RAMNIRANJAN JHUNJHUNWALA COLLEGE

GHATKOPAR (W), MUMBAI - 400 086

-

DEPARTMENT OF INFORMATION TECHNOLOGY

2020 - 2021

-

M.Sc.(I.T.) SEM I

Image and Vision Processing

-

Name:Surekha Omprakash Rajbhar

Roll No.: 12

Hindi Vidya Prachar Samiti's

RAMNIRANJAN JHUNJHUNWALA COLLEGE

(AUTONOMOUS)

Opposite Ghatkopar Railway Station, Ghatkopar West, Mumbai-400086

CERTIFICATE

This is to certify that Miss. **Surekha Omprakash Rajbhar** with Roll No. **12** has successfully completed the necessary course of experiments in the subject of **Image and Vision Processing** during the academic year **2020 – 2021** complying with the requirements of **RAMNIRANJAN JHUNJHUNWALA COLLEGE OF ARTS, SCIENCE AND COMMERCE**, for the course of **M.Sc. (IT)** semester -I.

_____ _____

Internal Examiner External Examiner

_____ _____

Head of Department College Seal

INDEX:

Practical 1

Implement Basic Intensity transformation functions

Image Inverse:

The negative or inverse of an image with intensity levels in the range [0, L-1] is obtained by using the negative transformation, which is given by the expression,

$S = L - 1 - r$

Where $L - 1$ (Maximum pixel value)

r (Pixel of an image)

In intensity, this means that the true black becomes true white and vise-versa.

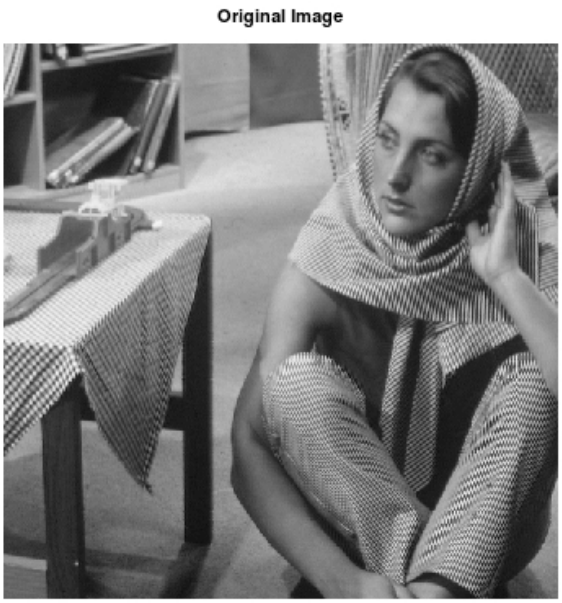
Reversing the intensity levels of an image in this manner produces the equivalent of a photographic negative. This type of processing is particularly suited for enhancing white or gray details embedded in dark regions of an image, especially when the black areas are dominant in size.

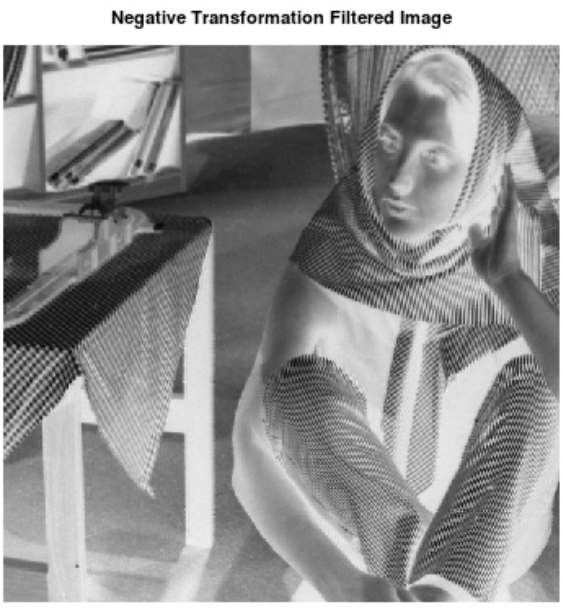
Program:

```
clear all;
close all;
pkg load image;

x = imread('barbara.png');   #Reading the image
#x = rgb2gray(x);            #Converting RGB image to gray-level
x = im2double(x);            #Converting image into double data type
[row col] = size(x);         #taking image size into matrix form
for i = 1:row   #reading row value from starting to end and storing in (i) variable
for j = 1:col     #reading row value from starting to end and storing in (j) variable
N(i,j)=1-x(i,j); #subtracting input matrix values from 255 and storing in new Matrix(N)
endfor
endfor
figure
imshow(x);
title('Original Image');
figure
imshow(N);
title('Negative Transformation Filtered Image');
```

Output:

```
clear all;
close all;
pkg load image;

x = imread('barbara.png');    #Reading the image
#x = rgb2gray(x);             #Converting RGB image to gray-level
x = im2double(x);             #Converting image into double data type
[row col] = size(x);          #taking image size into matrix form
for i = 1:row    #reading row value from starting to end and storing in (i) variable
for j = 1:col      #reading row value from starting to end and storing in (j) variable
N(i,j)=1-x(i,j); #subtracting input matrix values from 255 and storing in new Matrix(N)
endfor
endfor
figure
imshow(x);
title('Original Image');
figure
imshow(N);
title('Negative Transformation Filtered Image');
```





Roll No: 12

Practical 1(b)

Log Transformation:

The log transformation maps a narrow range of low intensity values in the input into a wider range of output levels. We use the transformation if this type to extend the values of dark pixel in an image while compress the higher-level values.

The general form of the log transformation is:

s = c log (r + 1)

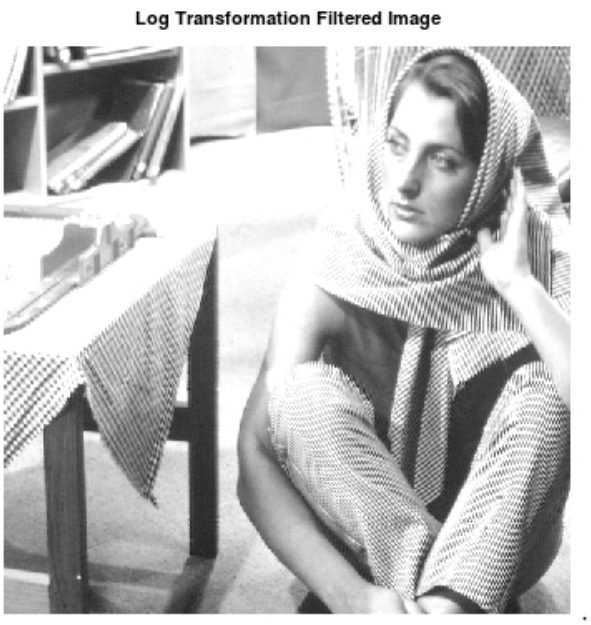
Where c is a constant, and $r \geq 0$.

Code:

```
x = imread('input.png');  #Reading the image
x = rgb2gray(x);          #Converting RGB image to gray-level image
x = im2double(x);         #Converting image into double data type
[row col] = size(x);      #taking image size into matrix form
c=2;                      #here we are taking constant value into c variable
for i = 1:row             #reading row value from starting to end and storing in (i) variable
for j = 1:col             #reading row value from starting to end and storing in (j) variable
N(i,j)=c*log(1+x(i,j));   #Here we are doing log calculation and storing the value into N
endfor
endfor
figure
imshow(x);
title('Original Image');
figure
imshow(N);
title('Log Transformation Filtered Image');
```

```
clear all;
close all;
pkg load image;
x = imread('barbara.png');  #Reading the image
#x = rgb2gray(x);              #Converting RGB image to gray-level image
x = im2double(x);            #Converting image into double data type
[row col] = size(x);          #taking image size into matrix form
c=2;                                  #here we are taking constant value into c variable
for i = 1:row                     #reading row value from starting to end and storing in (i)
for j = 1:col                       #reading row value from starting to end and storing in (j
N(i,j)=c*log(1+x(i,j));      #Here we are doing log calculation and storing the value into N
endfor
endfor
figure
imshow(x);
title('Original Image');
figure
imshow(N);
title('Log Transformation Filtered Image');
```

Output:



Roll No: 12

Practical No. 1 (c)

Power-Law Transformation:

Power-law curves with fractional values of γ map a narrow range of dark input values into a wider range of output values, with the opposite being true for higher values of input levels.

The nth power and nth root curves shown in below figure can be given by the expression as s = c r γ, This transformation function is also called as gamma correction. For various values of γ different levels of enhancements can be obtained. It is used to correct power law response phenomena. The different display monitors display images at different intensities and clarity. That means, every monitor has built-in gamma correction in it with certain gamma ranges and so a good monitor automatically corrects all the images displayed on it for the best contrast to give user the best experience. The gamma variation changes ratio of red green & blue along with intensity in color images. The difference between the log-transformation function and the power-law functions is that using the power-law function a family of possible transformation curves can be obtained just by varying the λ. This process is also called a gamma correction.

The Power Low Transformations can be given by the expression:

s = c * r ^ γ where,

s is the output pixels value

r is the input pixel value

c and γ are the real numbers

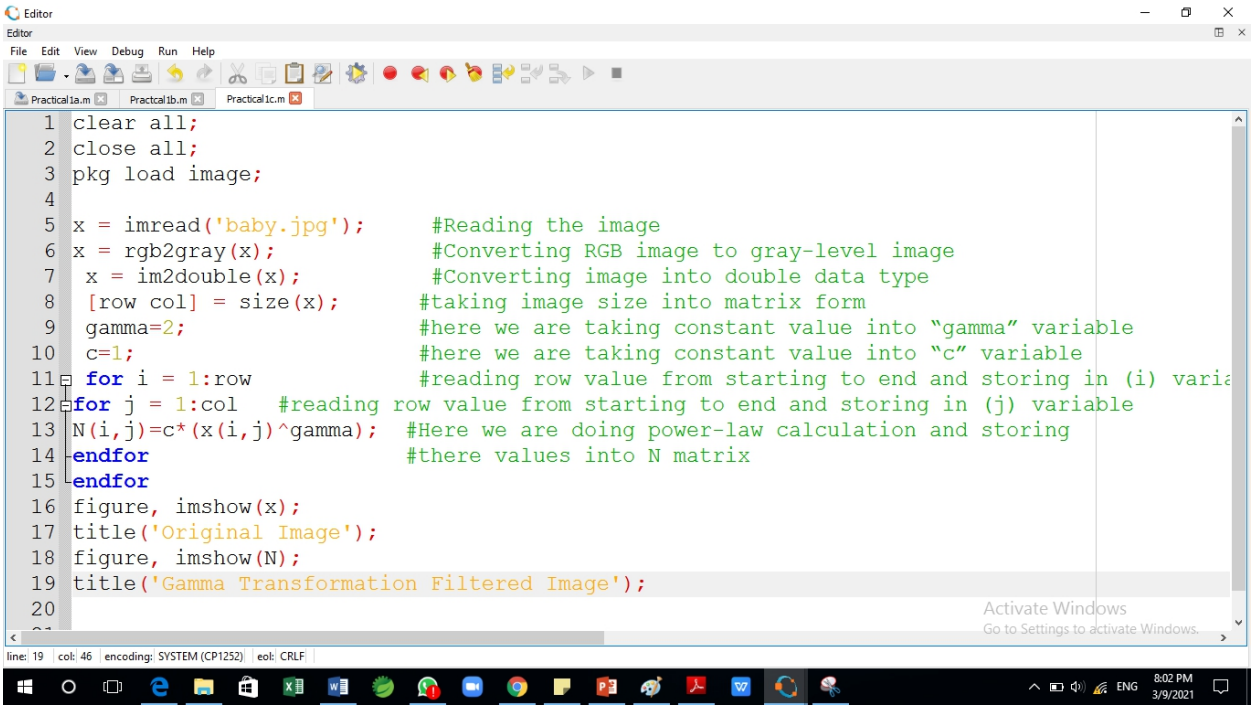
Code:

```
x = imread('baby.jpg'); #Reading the image
x = rgb2gray(x);          #Converting RGB image to gray-level image
 x = im2double(x);       #Converting image into double data type
 [row col] = size(x);    #taking image size into matrix form
 gamma=2;                #here we are taking constant value into “gamma” variable
 c=1;                    #here we are taking constant value into “c” variable
 for i = 1:row           #reading row value from starting to end and storing in (i) variable
        for j = 1:col    #reading row value from starting to end and storing in (j) variable
        N(i,j)=c*(x(i,j)^gamma);       #Here we are doing power-law calculation and storing .
        endfor                          there values into N matrix
endfor
figure, imshow(x);
```

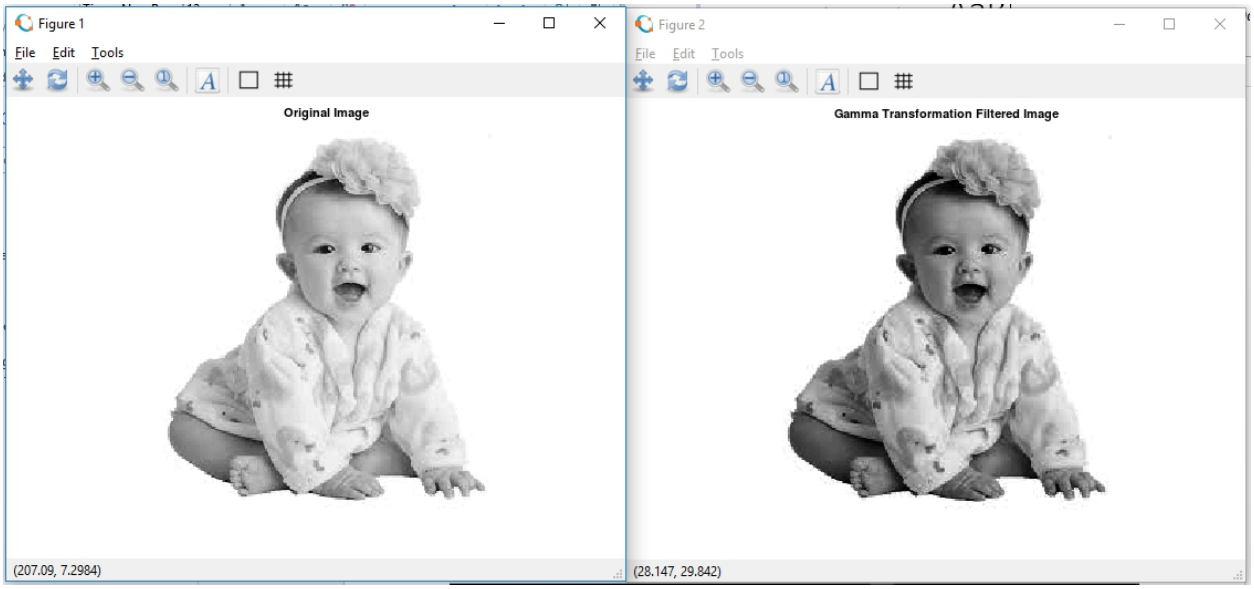
title('Original Image');

figure, imshow(N);

title('Gamma Transformation Filtered Image');



Output:



Roll No: 12

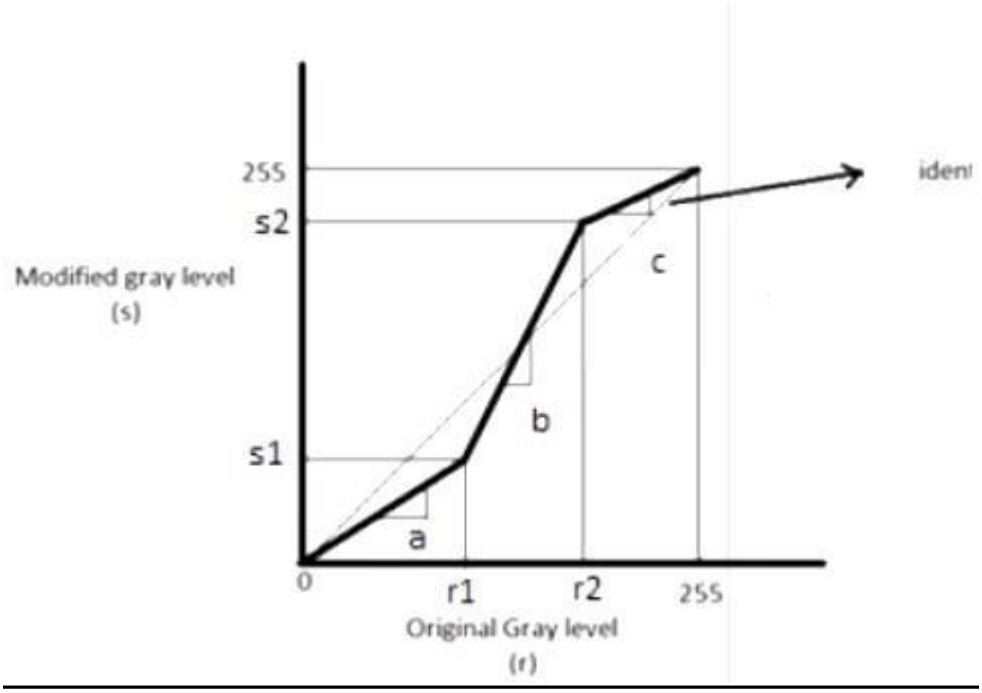
Practical No :- 2

[2]Picewise Transformation

[2.1]Contrast Streching

Contrast stretching (often called normalization) is a simple image enhancement technique that attempts to improve the contrast in an image by `stretching' the range of intensity values it contains to span a desired range of values, e.g. the full range of pixel values that the image type concerned allows.

It differs from the more sophisticated histogram equalization in that it can only apply a linear scaling function to the image pixel values. As a result the `enhancement' is less harsh.

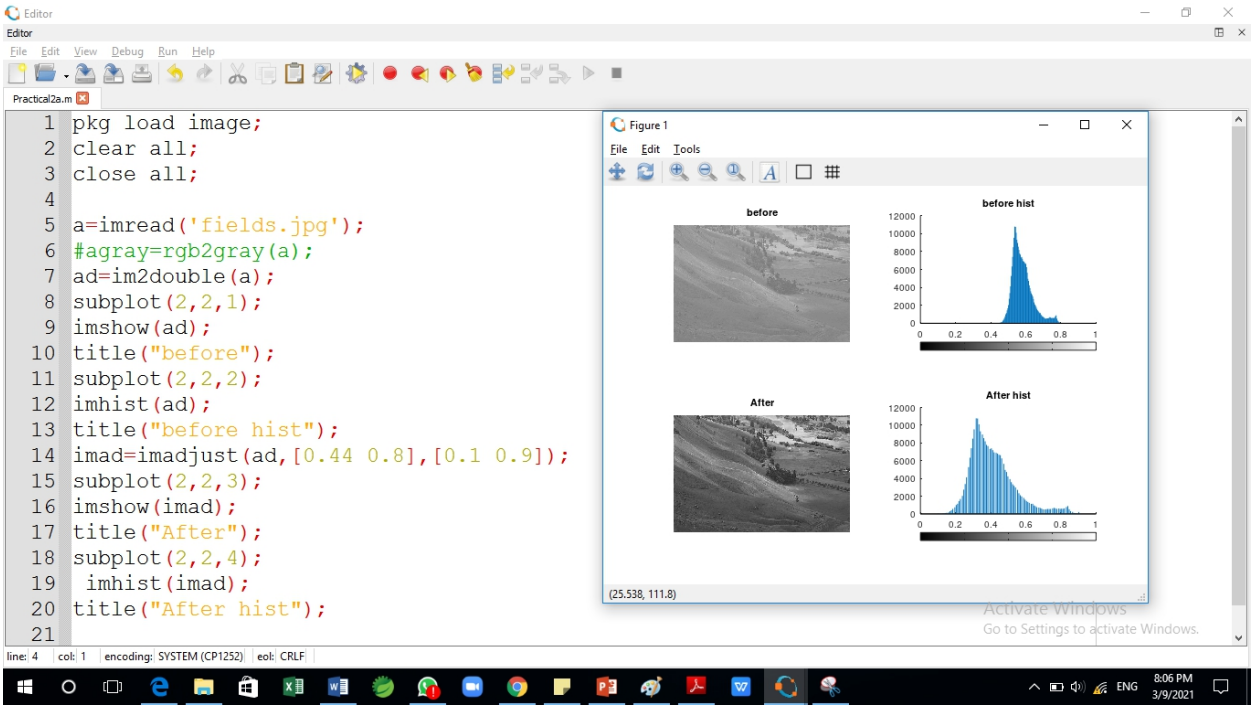


Using imadjust function-

```
pkg load image;
a=imread('fields.jpg');
#agray=rgb2gray(a);
ad=im2double(a);
subplot(2,2,1);
imshow(ad);
title("before");
```

```
subplot(2,2,2);
imhist(ad);
title("before hist");
imad=imadjust(ad,[0.44 0.8],[0.1 0.9]);
subplot(2,2,3);
imshow(imad);
title("After");
subplot(2,2,4);
 imhist(imad);
title("After hist");
```

Output:



Roll No: 12

Practical 2(a)- Contrast Stretching

Using inputs from user r1,r2,s1,s2

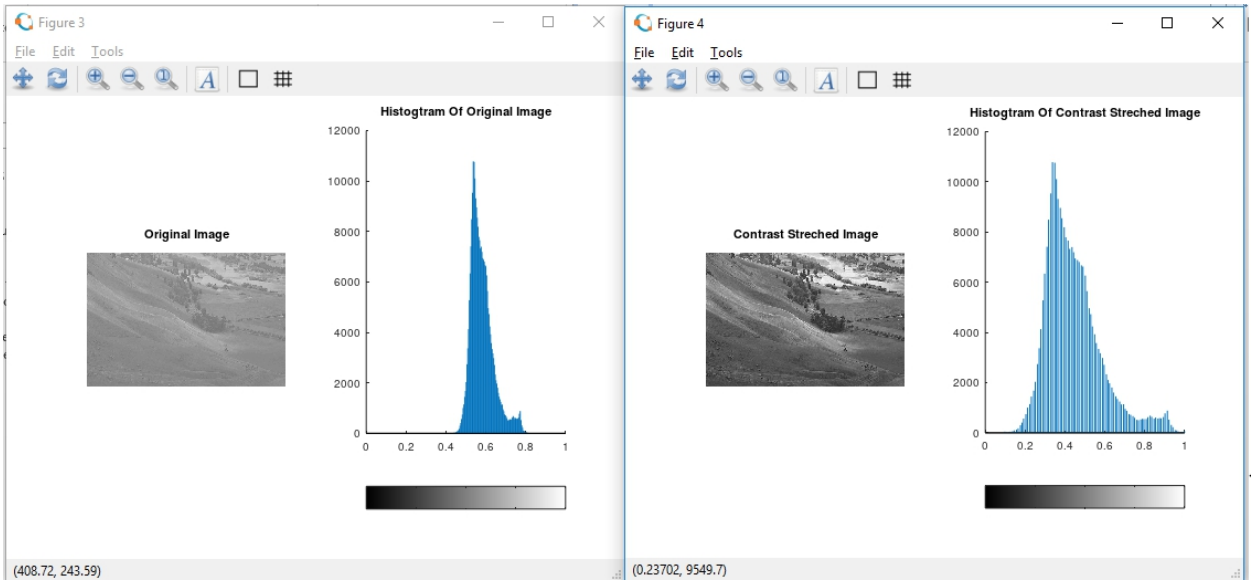
Program:

```
pkg load image;
clear all;
close all;
r = imread("fields.jpg");
#r=rgb2gray(r);
r = im2double(r);
[m n] = size(r);                % Getting the dimensions of the image.
#here we are taking 4 input from user
r1=input("Enter R1: ");
r2=input("Enter R2: ");
s1=input("Enter S1: ");
s2=input("Enter S2: ");
#Calculation of contrast stretching
a = s1/r1;
b = (s2-s1)/(r2-r1);
c = (255-s2)/(255-r2);
for i=1:m
  for j=1:n
    if r(i,j) < r1
    s(i,j) = a*r(i,j);
    elseif r(i,j) < r2
    s(i,j) = b*(r(i,j)-r1)+s1;
    else
    s(i,j) = c*(r(i,j)-r2)+s2;
    endif
  endfor
endfor
#Displaying the Original and Contrast Images
figure(3);
subplot(1,2,1)
imshow(r);
title("Original Image");
subplot(1,2,2)
imhist(r);
title('Histogtram Of Original Image');
figure(4);
subplot(1,2,1)
imshow(s);
title("Contrast Streched Image");
subplot(1,2,2)
imhist(s);
```

```
title('Histogtram Of Contrast Streched Image');
```

Output –

```
Enter R1: 0.44
Enter R2: 0.8
Enter S1: 0.1
Enter S2: 0.98
```



Roll No: 12

Practical 2(b)- Thresholding

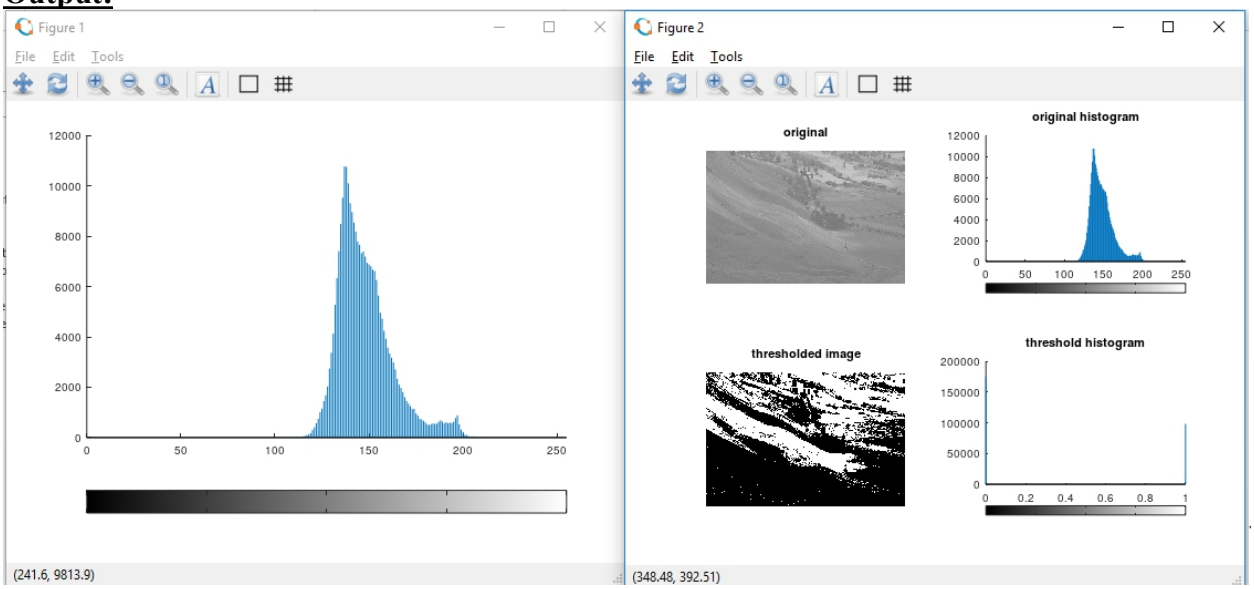
The input to a thresholding operation is typically a grayscale or color image. In the simplest implementation, the output is a binary image representing the segmentation. Black pixels correspond to background and white pixels correspond to foreground (or *vice versa*). In simple implementations, the segmentation is determined by a single parameter known as the *intensity threshold*. In a single pass, each pixel in the image is compared with this threshold. If the pixel's intensity is higher than the threshold, the pixel is set to, say, white in the output. If it is less than the threshold, it is set to black.

Code-

```
pkg load image;
clear all;
close all;

r=imread("fields.jpg");
#r=rgb2gray(r);
#r=im2double(r);
imhist(r);
thr=150;
[m n]=size(r);
s=zeros(m,n);
for i=1:m
  for j=1:n
    if(r(i,j)>thr)  #If dimension value is greater than user input then convert into 1
    s(i,j)=1;
    else              #If dimension value is less than user input then convert into 0
        s(i,j)=0;
    endif

  endfor
 endfor

#Displaying the Original Image with histogram
figure;

subplot(2,2,1); imshow(r); title("original");
subplot(2,2,2); imhist(r); title("original histogram");
subplot(2,2,3); imshow(s); title("thresholded image");
subplot(2,2,4); imhist(s); title("threshold histogram");
```

```
pkg load image;
clear all;
close all;

r=imread("fields.jpg");
#r=rgb2gray(r);
#r=im2double(r);
imhist(r);
thr=150;
[m n]=size(r);
s=zeros(m,n);
for i=1:m
  for j=1:n
    if(r(i,j)>thr)  #If dimension value is greater than user input then convert into 1
    s(i,j)=1;
    else                #If dimension value is less than user input then convert into 0
        s(i,j)=0;
    endif

  endfor
 endfor

#Displaying the Original Image with histogram
figure;

subplot(2,2,1); imshow(r); title("original");
subplot(2,2,2); imhist(r); title("original histogram");
subplot(2,2,3); imshow(s); title("thresholded image");
subplot(2,2,4); imhist(s); title("threshold histogram");
```

Output:



Roll No: 12

Practical -2 (C)

Bit plane slicing is well known technique used in Image processing. In image compression Bit plane slicing is used. Bit plane slicing is the conversion of image into multilevel binary image. These binary images are then compressed using different algorithm. With this technique, the valid bits from gray scale images can be separated, and it will be useful for processing these data in very less time complexity.
Digitally, an image is represented in terms of pixels. These pixels can be expressed further in terms of bits. Separating a digital image into its bit-planes is useful for analyzing the relative importance played by each bit of image, a process aids in determining the adequacy of the no. of bits used to quantize each pixel.

bitget:

C = bitget(A,bit)

C= bitget(A,bit) returns the value of the bit at position bit in A. Operand A must be an unsigned integer; a double or array containing unsigned integers, doubles or both. the bit input must be a number between 1 and the number of bits in the unsigned class of A. (e.g., 32 for the uint32 class).

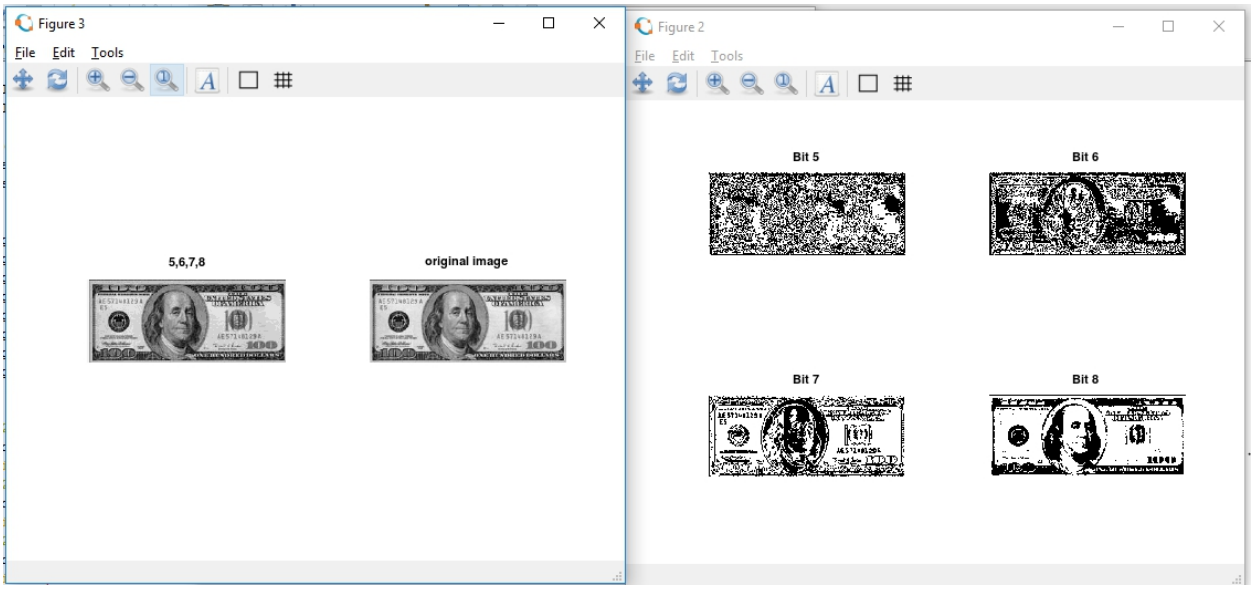
Image reconstruction using n bit planes.

Code-

```
pkg load image;
clear all;
close all;

A=imread('doller.png');
g=rgb2gray(A);
B=zeros(size(g));

#Getting the bit at specified position#
g1 = bitget(g,1);
g2 = bitget(g,2);
g3 = bitget(g,3);
g4 = bitget(g,4);
g5 = bitget(g,5);
g6 = bitget(g,6);
g7 = bitget(g,7);
g8 = bitget(g,8);

figure,
subplot(2,2,1)
imshow(logical(g1));
title('Bit 1');
subplot(2,2,2)
imshow(logical(g2));
title("Bit 2");
```

```
subplot(2,2,3)
imshow(logical(g3));
title('Bit 3');
subplot(2,2,4)
imshow(logical(g4));
title('Bit 4');

figure,
subplot(2,2,1)
imshow(logical(g5));
title('Bit 5');
subplot(2,2,2)
imshow(logical(g6));
title("Bit 6");
subplot(2,2,3)
imshow(logical(g7));
title('Bit 7');
subplot(2,2,4)
imshow(logical(g8));
title('Bit 8');

#B=bitset(B,4,bitget(A,4));
B=bitset(B,5,g5);
B=bitset(B,6,g6);
B=bitset(B,7,g7);
B=bitset(B,8,g8);
B=uint8(B);
figure,
subplot(1,2,1),imshow(B); title("5,6,7,8")
subplot(1,2,2),imshow(g); title("original image");
```

OUTPUT



1. The nth plane in the pixels are multiplied by the constant 2^n-1
2. For instance, consider the matrix
   A= A=[167 133 111
   144 140 135
   159 154 148] and the respective bit format

10100111	10000101	01101111
10010000	10001100	10000111
10011111	10011010	10010100

3. Combine the 8 bit plane and 7 bit plane.
   For 10100111, multiply the 8 bit plane with 128 and 7 bit plane with 64.\
   (1x128) + (0x64) + (1x0) + (0x0) + (0x0) + (1x0) + (1x0) + (1x0) = 128
4. Repeat this process for all the values in the matrix and the final result will be
   [128 128 64
   128 128 128
   128 128 128]

MATLAB CODE:

```
%Image reconstruction by combining 8 bit plane and 7 bit plane
A=imread('coins.png');
```

```
B=zeros(size(A));
B=bitset(B,7,bitget(A,7));
B=bitset(B,8,bitget(A,8));
B=uint8(B);
figure,imshow(B);

%Image reconstruction by combining 8,7,6 and 5 bit planes
A=imread('coins.png');
B=zeros(size(A));
B=bitset(B,8,bitget(A,8));
B=bitset(B,7,bitget(A,7));
B=bitset(B,6,bitget(A,6));
B=bitset(B,5,bitget(A,5));
B=uint8(B);
figure,imshow(B);
```

5. Explanation:

‘bitset’ is used to set a bit at a specified position. Use ‘bitget’ to get the bit at the positions 7 and 8 from all the pixels in matrix A and use ‘bitset’ to set these bit values at the positions 7 and 8 in the matrix B.

Practical No :- 3

Implement Histogram Equalization

Histogram modeling techniques (e.g. histogram equalization) provide a sophisticated method for modifying the dynamic range and contrast of an image by altering that image such that its intensity histogram has a desired shape. Unlike contrast stretching, histogram modeling operators may employ non-linear and non-monotonic transfer functions to map between pixel intensity values in the input and output images. Histogram equalization employs a monotonic, non-linear mapping which re-assigns the intensity values of pixels in the input image such that the output image contains a uniform distribution of intensities (i.e. a flat histogram). This technique is used in image comparison processes (because it is effective in detail enhancement) and in the correction of non-linear effects introduced by, say, a digitizer or display system.

The process of adjusting intensity values can be done automatically using histogram equalization. Histogram equalization involves transforming the intensity values so that the histogram of the output image approximately matches a specified histogram. By default, the histogram equalization function, histeq, tries to match a flat histogram with 64 bins, but we can specify a different histogram instead.

Syntax: X=histeq(y);

histeq enhances the contrast of images by transforming the values in an intensity image, or the values in the colormap of an indexed image, so that the histogram of the output image approximately matches a specified histogram.

Histogram equalization without using histeq() function.

The following steps are performed to obtain histogram equalization:

1. Find the frequency of each pixel value.
   Consider a matrix A= $\begin{bmatrix} 1 & 4 & 2 \\ 5 & 1 & 3 \\ 1 & 2 & 4 \end{bmatrix}$ with no of bins =5.
   The pixel value 1 occurs 3 times.
   Similarly the pixel value 2 occurs 2 times and so on.
2. Find the probability of each frequency.
   The probability of pixel value 1's occurrence = frequency (1)/no of pixels.
   i.e 3/9.
3. Find the cumulative histogram of each pixel:
   The cumulative histogram of 1 = 3.
   Cumulative histogram of 2 = cumulative histogram of 1 + frequency of 2=5.
   Cumulative histogram of 3 =
   cumulative histogram of 2+frequency of 3 = 5+1=6.
4. Find the cumulative distribution probability of each pixel
   cdf of 1= cumulative histogram of 1/no of pixels= 3/9.
5. Calculate the final value of each pixel by multiplying cdf with (no of bins);
   cdf of 1= (3/9)*(5)=1.6667. Round off the value.
6. Now replace the final values : $\begin{bmatrix} (2) & 4 & 3 \\ 5 & (2) & 3 \\ (2) & 3 & 4 \end{bmatrix}$

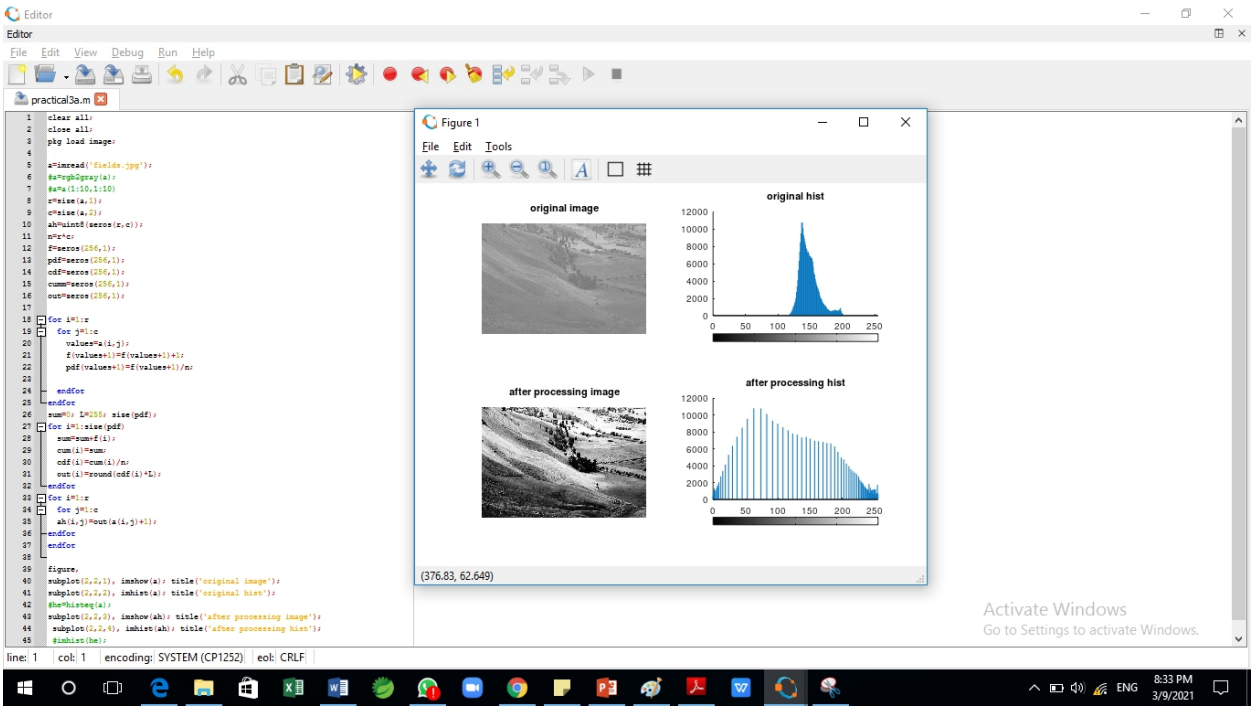
The final value for bin 1 is 2. It is placed in the place of 1 in the matrix.

Code:

```
clear all;
close all;
pkg load image;

a=imread('fields.jpg');
#a=rgb2gray(a);
#a=a(1:10,1:10)
r=size(a,1);
c=size(a,2);
ah=uint8(zeros(r,c));
n=r*c;
f=zeros(256,1);
pdf=zeros(256,1);
cdf=zeros(256,1);
cumm=zeros(256,1);
out=zeros(256,1);

for i=1:r
  for j=1:c
    values=a(i,j);
    f(values+1)=f(values+1)+1;
    pdf(values+1)=f(values+1)/n;

  endfor
endfor
sum=0; L=255; size(pdf);
for i=1:size(pdf)
  sum=sum+f(i);
  cum(i)=sum;
  cdf(i)=cum(i)/n;
```

```
  out(i)=round(cdf(i)*L);
endfor
for i=1:r
  for j=1:c
  ah(i,j)=out(a(i,j)+1);
endfor
endfor

figure,
subplot(2,2,1), imshow(a); title('original image');
subplot(2,2,2), imhist(a); title('original hist');
#he=histeq(a);
subplot(2,2,3), imshow(ah); title('after processing image');
 subplot(2,2,4), imhist(ah); title('after processing hist');
 #imhist(he);
```

Output:



Roll No: 12

Practical -4

Low-pass Filter/Smoothing

Filters (Average, Weighted Average and Median)

Low pass filtering (smoothing), is employed to remove high spatial frequency noise from a digital image. The low-pass filters usually employ moving window operator which affects one pixel of the image at a time, changing its value by some function of a local region (window) of pixels. The operator moves over the image to affect all the pixels in the image.

Average Filter

Average or mean filtering is easy to implement. It is used as a method of smoothing images, reducing the amount of intensity variation between one pixel and the next resulting in reducing noise in images.

The idea of average filtering is simply to replace each pixel value in an image with the mean (`average') value of its neighbours, including itself. This has the effect of eliminating pixel values which are unrepresentative of their surroundings. Mean filtering is usually thought of as a convolution filter. Like other convolutions it is based around a kernel, which represents the shape and size of the neighbourhood to be sampled when calculating the mean. Often a 3×3 square kernel is used, as shown below:

$$\frac{1}{9}\begin{bmatrix}1 & 1 & 1\\ 1 & 1 & 1\\ 1 & 1 & 1\end{bmatrix}$$

Syntax:

```
img = imread ('hawk.png');
mf = ones(3,3)/9;
```

The mf is the mean filter:

```
>> mf = ones(3,3)/9
mf =
    0.1111 0.1111 0.1111
    0.1111 0.1111 0.1111
    0.1111 0.1111 0.1111
```

The filter2() is defined as:

Y = filter2(h,X) filters the data in X with the two-dimensional FIR filter in the matrix h. It computes the result, Y, using two-dimensional correlation, and returns the central part of the correlation that is the same size as X.

Y = filter2(h,X,shape)

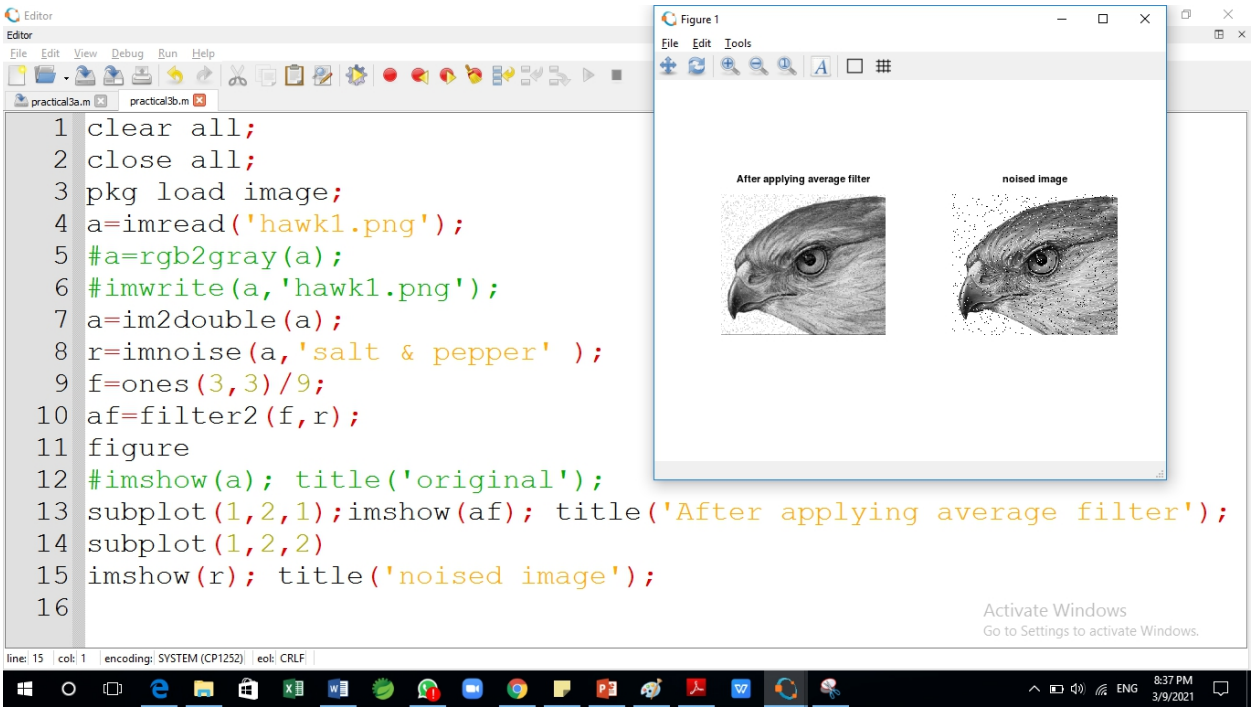
It returns the part of Y specified by the shape parameter. shape is a string with one of these values:

'**full**' : Returns the full two-dimensional correlation. In this case, Y is larger than X.
'**same**' : (default) Returns the central part of the correlation. In this case, Y is the same size as X.
'**valid**' : Returns only those parts of the correlation that are computed without zero-padded edges. In this case, Y is smaller than X.

Code:

```
clear all;
close all;
pkg load image;
a=imread('hawk1.png');
#a=rgb2gray(a);
#imwrite(a,'hawk1.png');
a=im2double(a);
r=imnoise(a,'salt & pepper' );
f=ones(3,3)/9;
af=filter2(f,r);
figure
#imshow(a); title('original');
subplot(1,2,1);imshow(af); title('After applying average filter');
subplot(1,2,2)
imshow(r); title('noised image');
```

Output:


Editor
File Edit View Debug Run Help
practical3a.m
practical3b.m
1 clear all;
2 close all;
3 pkg load image;
4 a=imread('hawk1.png');
5 #a=rgb2gray(a);
6 #imwrite(a,'hawk1.png');
7 a=im2double(a);
8 r=imnoise(a,'salt & pepper');
9 f=ones(3,3)/9;
10 af=filter2(f,r);
11 figure
12 #imshow(a); title('original');
13 subplot(1,2,1);imshow(af); title('After applying average filter');
14 subplot(1,2,2)
15 imshow(r); title('noised image');
16
Figure 1
File Edit Tools
After applying average filter
noised image
Activate Windows
Go to Settings to activate Windows.
line: 15 col: 1 encoding: SYSTEM (CP1252) eol: CRLF
ENG
8:37 PM
3/9/2021

Roll No: 12

Practical-4 (Low pass- Average filter without using inbuilt functions)

Weighted Average

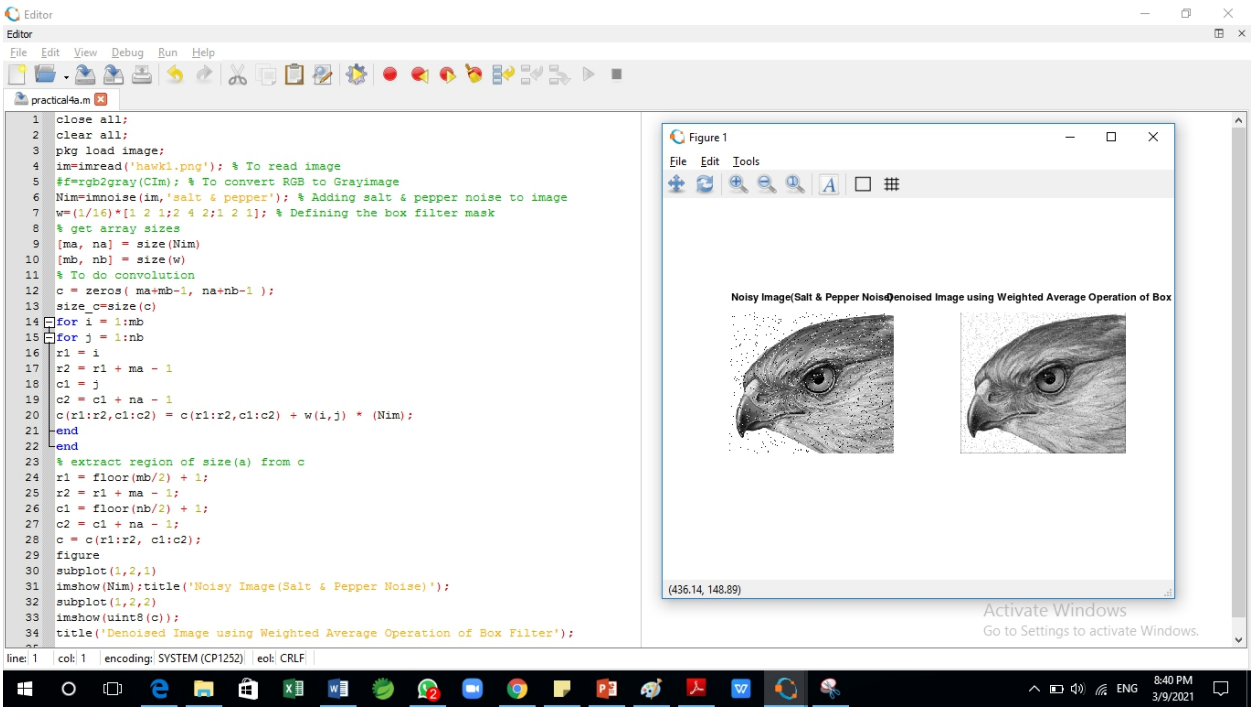
The second mask is a little more interesting. This mask yields a so-called weighted average, terminology used to indicate that pixels are multiplied by different coefficients, thus giving more importance (weight) to some pixels at the expense of others. In the mask the pixel at the center of the mask is multiplied by a higher value than any other, thus giving this pixel more importance in the calculation of the average.
Weighted Filter mask is as follows:

Code:

```
close all;

clear all;

pkg load image;

im=imread('hawk1.png'); % To read image

#f=rgb2gray(CIm); % To convert RGB to Grayimage

Nim=imnoise(im,'salt & pepper'); % Adding salt & pepper noise to image

w=(1/16)*[1 2 1;2 4 2;1 2 1]; % Defining the box filter mask

% get array sizes

[ma, na] = size(Nim)

[mb, nb] = size(w)


% To do convolution

c = zeros( ma+mb-1, na+nb-1 );

size_c=size(c)

for i = 1:mb

for j = 1:nb

r1 = i

r2 = r1 + ma - 1

c1 = j

c2 = c1 + na - 1


c(r1:r2,c1:c2) = c(r1:r2,c1:c2) + w(i,j) * (Nim);

end
```

```
end
% extract region of size(a) from c
r1 = floor(mb/2) + 1;
r2 = r1 + ma - 1;
c1 = floor(nb/2) + 1;
c2 = c1 + na - 1;
c = c(r1:r2, c1:c2);
figure
subplot(1,2,1)
imshow(Nim);
title('Noisy Image(Salt & Pepper Noise)');
subplot(1,2,2)
imshow(uint8(c));
title('Denoised Image using Weighted Average Operation of Box Filter');
```

Output:



Roll No: 12

Practical No- 4

Low Pass Filter – (Median Filter)

#Median Spacial Domain Filtering

Median
The best-known example in this category is the median filter, which, as its name implies, replaces the value of a pixel by the median of the gray levels in the neighborhood of that pixel (the original value of the pixel is included in the computation of the median). Median filters are quite popular because, for certain types of random noise, they provide excellent noise-reduction capabilities, with considerably less blurring than linear smoothing filters of similar size. Median filters are particularly effective in the presence of impulse noise, also called salt-and-pepper noise because of its appearance as white and black dots superimposed on an image.
How It work:
Like the mean filter, the median filter considers each pixel in the image in turn and looks at its nearby neighbors to decide whether or not it is representative of its surroundings. Instead of simply replacing the pixel value with the *mean* of neighboring pixel values, it replaces it with the *median* of those values. The median is calculated by first sorting all the pixel values from the surrounding neighborhood into numerical order and then replacing the pixel being considered with the middle pixel value. (If the neighborhood under consideration contains an even number of pixels, the average of the two middle pixel values is used.)

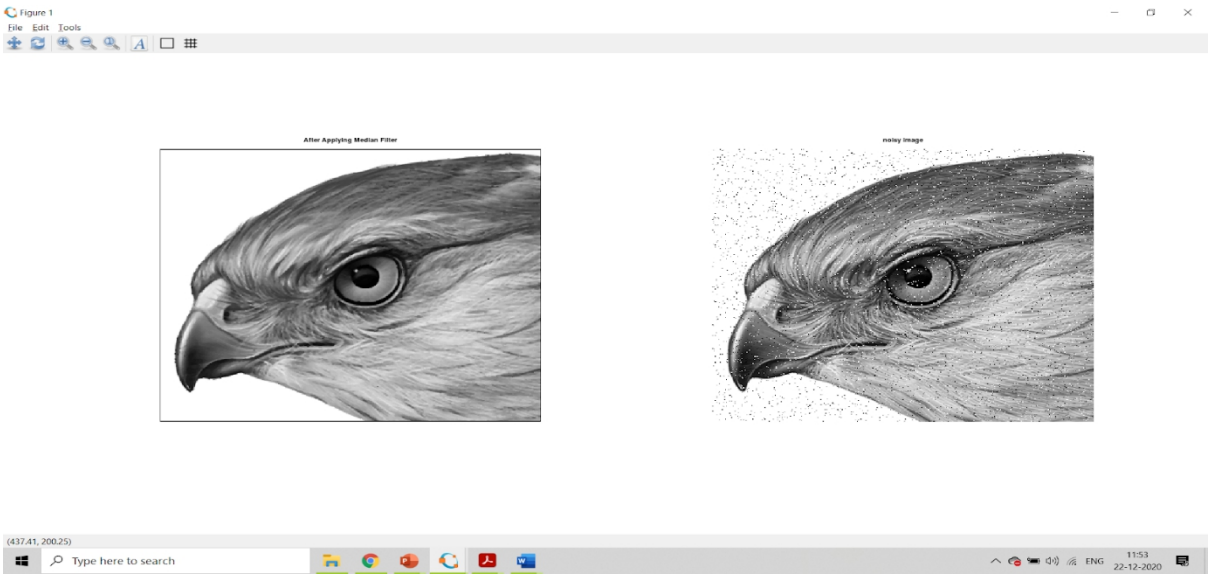
Syntax:
B = medfilt2(A) performs median filtering of the matrix A using the default 3-by-3 neighborhood

Code:

```
pkg load image;

# Read the image

 a=imread('hawk1.png');

 img_noisy1=imnoise(a,'salt & pepper' );

# Obtain the number of rows and columns of the image

[m, n] = size(img_noisy1)

# Traverse the image. For every 3X3 area,

# find the median of the pixels and

# replace the center pixel by the median

img_new1 = zeros(m, n);


for i=2: m-1

    for j =2: n-1
```

```
        temp = [img_noisy1(i-1, j-1),
                img_noisy1(i-1, j),
                img_noisy1(i-1, j + 1),
                img_noisy1(i, j-1),
                img_noisy1(i, j),
                img_noisy1(i, j + 1),
                img_noisy1(i + 1, j-1),
                img_noisy1(i + 1, j),
                img_noisy1(i + 1, j + 1)] ;
        temp = sort(temp);
        img_new1(i, j)= temp(4);
  endfor
  endfor
img_new1 = uint8(img_new1);
figure
subplot(1,2,1); imshow(img_new1); title('After Applying Median Filter');
subplot(1,2,2); imshow(img_noisy1);title('noisy image');
```

Output:



Roll No: 12

Second order derivative – The Laplacian Filter

The Laplacian is a 2-D isotropic measure of the 2nd spatial derivative of an image. The Laplacian of an image highlights regions of rapid intensity change and is therefore often used for edge detection.
The Laplacian is often applied to an image that has first been smoothed with something approximating a Gaussian smoothing filter in order to reduce its sensitivity to noise, and hence the two variants will be described together here. The operator normally takes a single gray-level image as input and produces another gray-level image as output.
How It Work:

$$L(x,y) = \frac{\partial^2 I}{\partial x^2} + \frac{\partial^2 I}{\partial y^2}$$

The Laplacian L(x,y) of an image with pixel intensity values I(x,y) is given by:
This can be calculated using a convolution filter.
Since the input image is represented as a set of discrete pixels, we have to find a discrete convolution kernel that can approximate the second derivatives in the definition of the Laplacian. Two commonly used small kernels are shown in below.

0	−1	0
−1	4	−1
0	−1	0

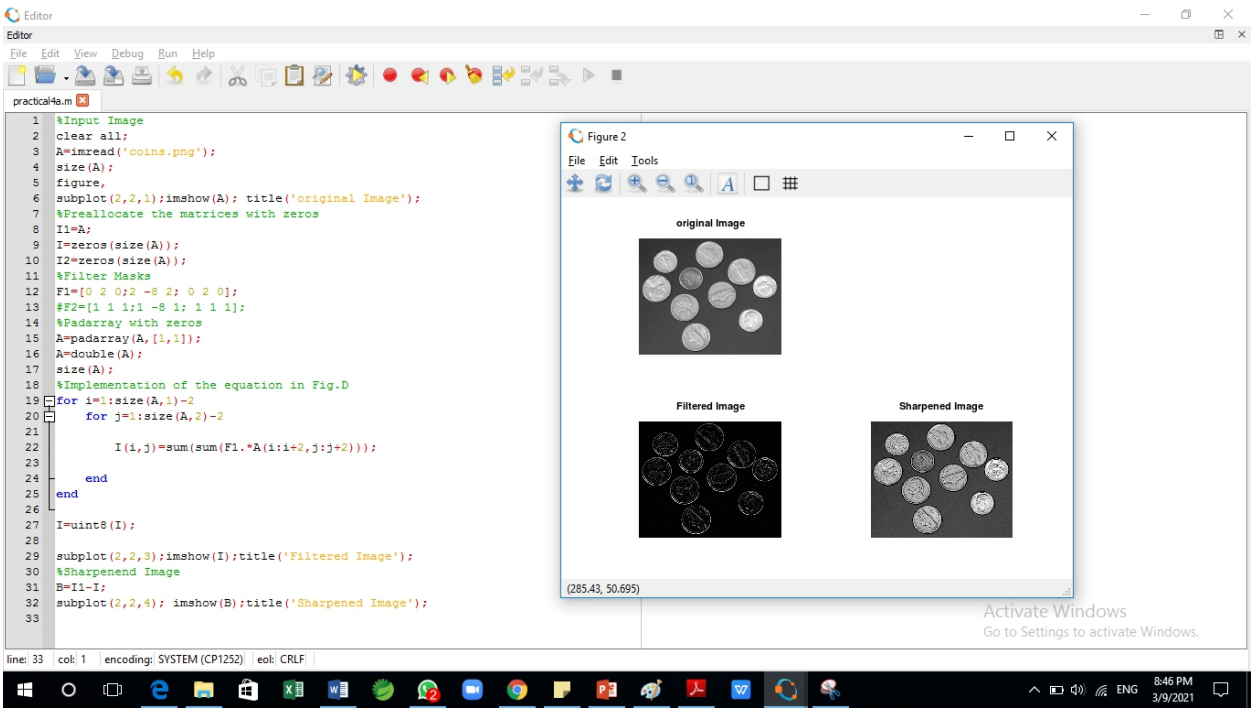
−1	−1	−1
−1	8	−1
−1	−1	−1

Code:

```
%Input Image
clear all;
A=imread('coins.png');
size(A);
figure,
subplot(2,2,1);imshow(A); title('original Image');
%Preallocate the matrices with zeros
I1=A;
I=zeros(size(A));
I2=zeros(size(A));
%Filter Masks
F1=[0 2 0;2 -8 2; 0 2 0];
#F2=[1 1 1;1 -8 1; 1 1 1];
%Padarray with zeros
A=padarray(A,[1,1]);
```

```
A=double(A);
size(A);
%Implementation of the equation in Fig.D
for i=1:size(A,1)-2
   for j=1:size(A,2)-2

       I(i,j)=sum(sum(F1.*A(i:i+2,j:j+2)));

   end
end

I=uint8(I);


subplot(2,2,3);imshow(I);title('Filtered Image');
%Sharpenend Image
B=I1-I;
subplot(2,2,4); imshow(B);title('Sharpened Image');
```

Output-



Roll No: 12

First Order Derivative -Sobel Operator for edge detection without using edge function

Code:

```
clear all;
A=imread('peppers.png');
figure,
subplot(1,2,1); imshow(A); title('Original');
C=double(A);
size(C)

for i=1:size(C,1)-2
   for j=1:size(C,2)-2
      %Sobel mask for x-direction:
      Gx=((C(i+2,j)+2*C(i+2,j+1)+C(i+2,j+2))-(C(i,j)+2*C(i,j+1)+C(i,j+2)));
      %Sobel mask for y-direction:
      Gy=((C(i,j+2)+2*C(i+1,j+2)+C(i+2,j+2))-(C(i,j)+2*C(i+1,j)+C(i+2,j)));
      %The gradient of the image
      # B(i,j)=abs(Gx)+abs(Gy);
      A(i,j)=sqrt(Gx.^2+Gy.^2);

   end
end
subplot(1,2,2); imshow(A); title('Sobel gradient');
```

First Order Derivative -Sobel Operator for edge detection using edge function

```
#load package of image
pkg load image;
#Take input image
img=imread("peppers.png");

#function to find edge using sobel filter
sobel = edge(img,'Sobel');
figure 1,
subplot(1,2,1)
imshow(img);
title('Original Image');
subplot(1,2,2)
imshow(sobel);
title("Edge detection using sobel filter");
#function to find edge using sobel filter
robert = edge(img,'Roberts');
prewitt = edge(img,'Prewitt');
figure 2,
subplot(1,2,1)
imshow(robert);
title('Edge detection using robert filter');
subplot(1,2,2)
imshow(prewitt);
title("Edge detection using prewitt filter");
```


Figure 1
File Edit Tools
Original
Sobel gradient
(176.53, 472.27)
Type here to search
ENG
10:54
24-12-2020

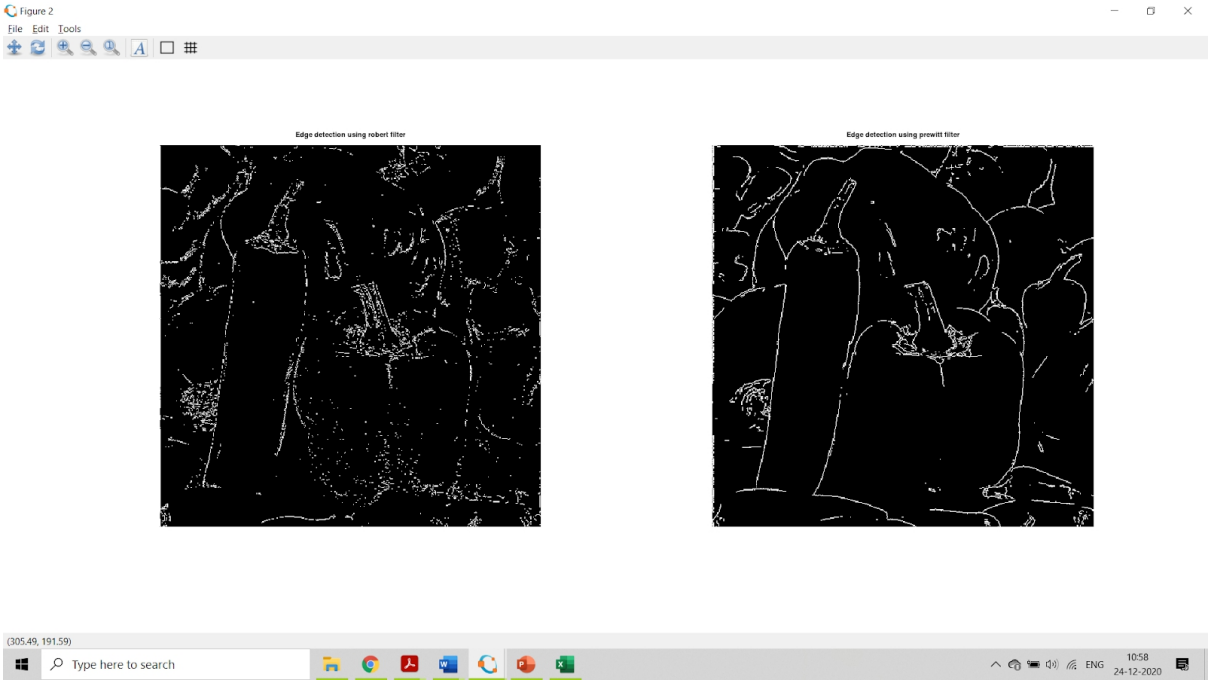


Figure 2
File Edit Tools
Edge detection using robert filter
Edge detection using prewitt filter
(305.49, 191.59)
Type here to search
ENG
10:58
24-12-2020

Roll No: 12


Figure 1
File Edit Tools
Original Image
Edge detection using sobel filter
(42.511, 51.676)
Type here to search
ENG
10:57
24-12-2020

Roll No: 12

Practical no. 6 Color Image Processing

Human vision is more sensitive to color than gray levels. Therefore, color image processing is important, although it requires more memory to store and longer execution times to process. There are different color models, and each one is suitable for some application. In the RGB model, a color image is expressed in terms of the intensities of its red, green, and blue components.
The human visual system can distinguish hundreds of thousands of different colour shades and intensities, but only around 100 shades of grey.
Therefore, in an image, a great deal of extra information may be contained in the colour, and this extra information can then be used to simplify image analysis, e.g. object identification and extraction based on colour.
Color Image processing is divided into three types:-

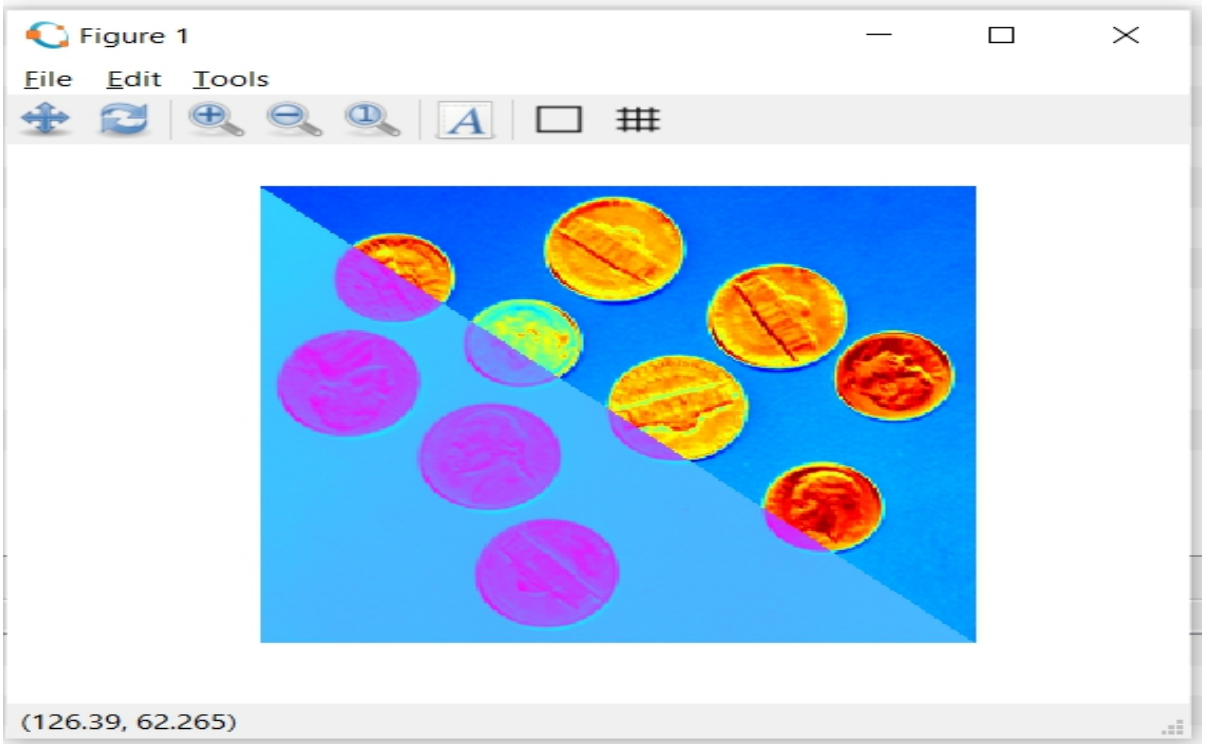
D. A) Pseudocoloring

Pseudo-color processing is a technique that maps each of the grey levels of a black and white image into an assigned color. This colored image, when displayed, can make the identification of certain features easier for the observer. The mappings are computationally simple and fast **Pseudo-color** schemes can also be designed to preserve or remove intensity information.

Code:

```
pkg load image;

close all;

clear all;

%READ INPUT IMAGE

A = imread('coins.png');

%RESIZE IMAGE

A = imresize(A,[256 256]);

%PRE-ALLOCATE THE OUTPUT MATRIX

Output = ones([size(A,1) size(A,2)]);


%COLORMAPS

#maps={'jet(256)';'hsv(256)';'cool(256)';'spring(256)';'summer(256)';'parula(256)';'hot(256)'};

%COLORMAP 1

map = colormap(jet(256));

Red = map(:,1);

Green = map(:,2);

Blue = map(:,3);
```

```
R1 = Red(A);
G1 = Green(A);
B1 = Blue(A);

%COLORMAP 2
map = colormap(cool(256));
Red = map(:,1);
Green = map(:,2);
Blue = map(:,3);

%RETRIEVE POSITION OF UPPER TRIANGLE
[x,y]=find(triu(Output)==1);

Output(:,:,1) = Red(A);
Output(:,:,2) = Green(A);
Output(:,:,3) = Blue(A);
for i=1:numel(x)
    Output(x(i),y(i),1)=R1(x(i),y(i));
    Output(x(i),y(i),2)=G1(x(i),y(i));
    Output(x(i),y(i),3)=B1(x(i),y(i));
end
Output = im2uint8(Output);
%FINAL IMAGE
imshow(Output);
```



Code – Intensity slicing

```
clear all;
#im=input('Enter the file name);
input_image=imread('hawk.png');
k=rgb2gray(input_image);
[x y z]=size(k);
% z should be one for the input image
k=double(k);
for i=1:x
for j=1:y
if k(i,j)>=0 && k(i,j)<50
m(i,j,1)=k(i,j,1)+25;
m(i,j,2)=k(i,j)+50;
m(i,j,3)=k(i,j)+60;
end
```

```
if k(i,j)>=50 && k(i,j)<100
m(i,j,1)=k(i,j)+55;
m(i,j,2)=k(i,j)+68;
m(i,j,3)=k(i,j)+70;
end
if k(i,j)>=100 && k(i,j)<150
m(i,j,1)=k(i,j)+52;
m(i,j,2)=k(i,j)+30;
m(i,j,3)=k(i,j)+15;
end
if k(i,j)>=150 && k(i,j)<200
m(i,j,1)=k(i,j)+50;
m(i,j,2)=k(i,j)+40;
m(i,j,3)=k(i,j)+25;
end
if k(i,j)>=200 && k(i,j)<=256
m(i,j,1)=k(i,j)+120;
m(i,j,2)=k(i,j)+60;
m(i,j,3)=k(i,j)+45;
end
end
end
figure,
imshow(uint8(k),[]);
title('Original Image');
figure,
imshow(uint8(m),[]);
title("Pseudo filtred image");
```

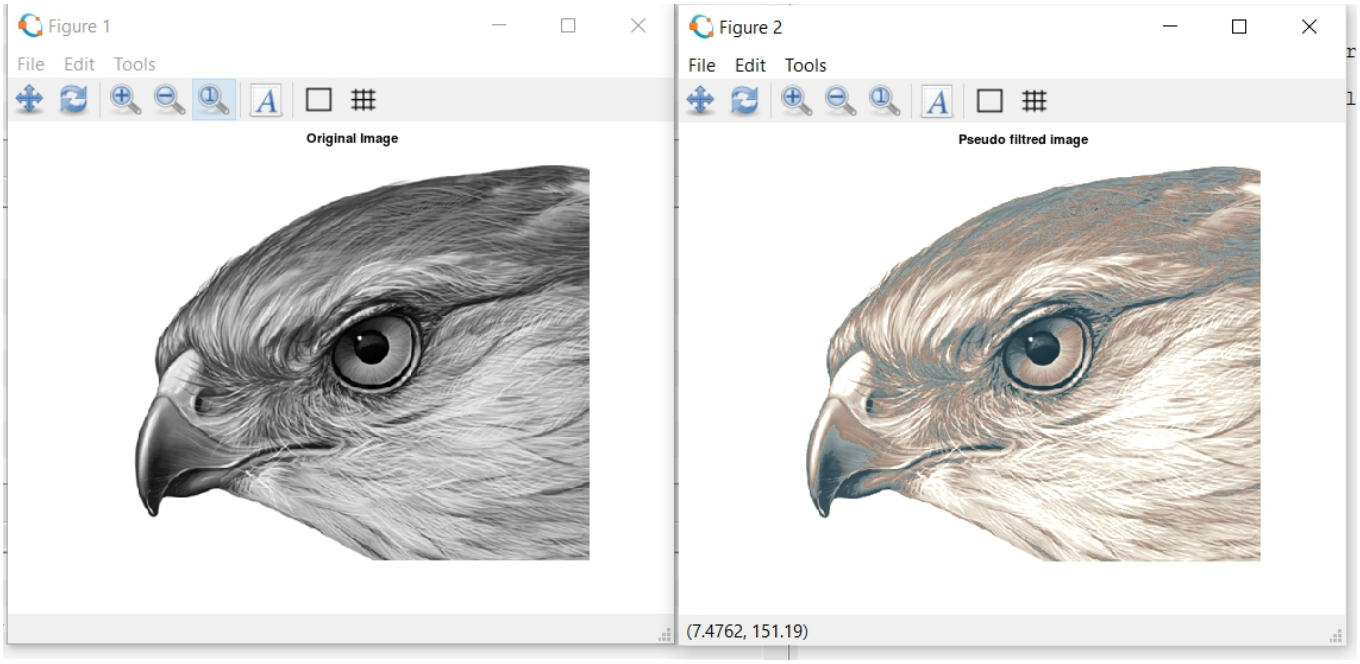


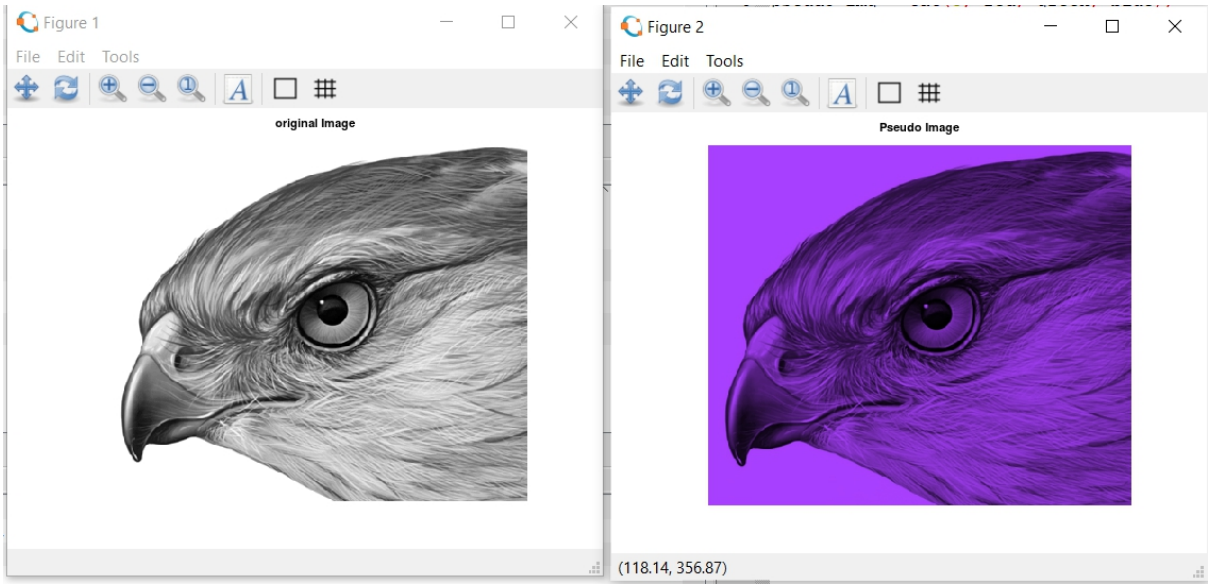
Figure 1
File Edit Tools
Original Image
Figure 2
File Edit Tools
Pseudo filtred image
(7.4762, 151.19)

Roll No: 12

Pseudo Image:-

```
pkg load image; clear all;
img = imread('hawk1.png'); % Read image
figure, imshow(img);title("original Image");
red = 0.66*img;
green=0.25*img;
blue = img;
pseudo_img = cat(3, red, green, blue);
figure, imshow(pseudo_img), title('Pseudo Image');
```

Output:



Roll No: 12

Practical No :-7

Image Compression techniques and Watermarking

Image Compression is a technique of reducing amount of data required to represent data.
It involves :
Reducing the storage required to save a image
Reducing the bandwidth reqired to transmit it

[7.1]Implement Huffman coding

Huffman code is a type of optimal prefix code that is commonly used for **lossless** data compression. The algorithm has been developed by David A. Huffman.
The main idea is to transform plain input into variable-length code. As in other entropy encoding methods, more common symbols are generally represented using **fewer bits** than less common symbols. The easiest way to understand how to create Huffman tree is to analyze following steps:
1. Scan text for symbols (e.g. 1-byte characters) and calculate their **frequency** of occurrence. Symbol value with its count of occurrences is a single leaf.

2. Start a loop.

3. Find two smallest probability nodes and **combine them** into single node.

4. Remove those two nodes from list and **insert** combined one.

5. Repeat the loop until the list has only single node.

6. This last single node represent a huffman tree.

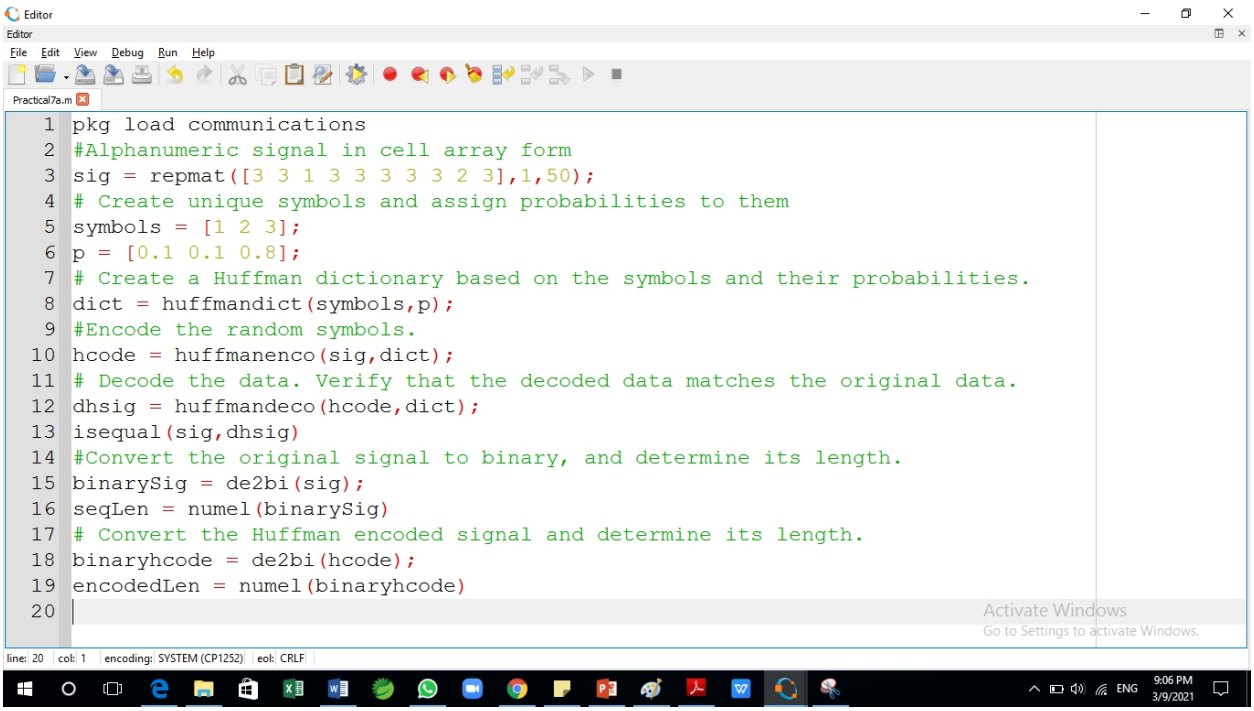
Huffmanenco. This function is used in Huffman coding. The syntax is:
hcode = huffmanenco(sig,dict)
This line encodes the signal „sig‟ described by the „dict‟ dictionary. The argument „sig‟ can have the form of a numeric vector, numeric cell array or alphanumeric cell array. If „sig‟ is a cell array, it must be either a row or a column. The „dict‟ is an Nx2 cell array, where „N‟ is the number of distinct
possible symbols to be encoded. The first column of „dict‟ represents the distinct symbols and the second column represents the corresponding codewords.

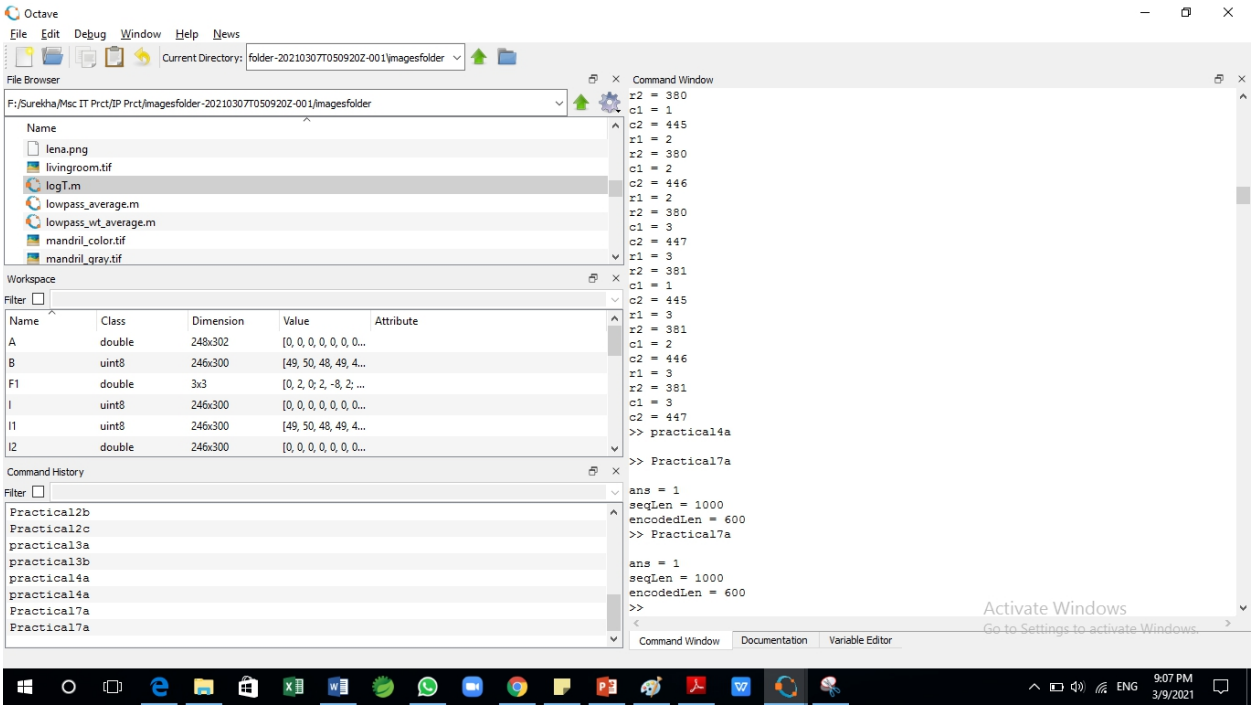
Huffmandeco. This function is used in Huffman decoding. The syntax is:
dsig = huffmandeco(comp,dict)
This line decodes the numeric Huffman code vector comp using the code dictionary „dict‟. The argument „dict‟ is an Nx2 cell array, where „N‟ is the number of distinct possible symbols in the original signal that was encoded as „comp‟. The first column of „dict‟ represents the distinct symbols and the second column represents the corresponding codewords.

Implement Huffman Coding

Code:

```
pkg load communications
#Alphanumeric signal in cell array form
sig = repmat([3 3 1 3 3 3 3 3 2 3],1,50);
# Create unique symbols and assign probabilities to them
symbols = [1 2 3];
p = [0.1 0.1 0.8];
# Create a Huffman dictionary based on the symbols and their probabilities.
dict = huffmandict(symbols,p);
#Encode the random symbols.
hcode = huffmanenco(sig,dict);
# Decode the data. Verify that the decoded data matches the original data.
dhsig = huffmandeco(hcode,dict);
isequal(sig,dhsig)
#Convert the original signal to binary, and determine its length.
binarySig = de2bi(sig);
seqLen = numel(binarySig)
# Convert the Huffman encoded signal and determine its length.
binaryhcode = de2bi(hcode);
encodedLen = numel(binaryhcode)
```


Editor
File Edit View Debug Run Help
Practical7a.m
1 pkg load communications
2 #Alphanumeric signal in cell array form
3 sig = repmat([3 3 1 3 3 3 3 3 2 3],1,50);
4 # Create unique symbols and assign probabilities to them
5 symbols = [1 2 3];
6 p = [0.1 0.1 0.8];
7 # Create a Huffman dictionary based on the symbols and their probabilities.
8 dict = huffmandict(symbols,p);
9 #Encode the random symbols.
10 hcode = huffmanenco(sig,dict);
11 # Decode the data. Verify that the decoded data matches the original data.
12 dhsig = huffmandeco(hcode,dict);
13 isequal(sig,dhsig)
14 #Convert the original signal to binary, and determine its length.
15 binarySig = de2bi(sig);
16 seqLen = numel(binarySig)
17 # Convert the Huffman encoded signal and determine its length.
18 binaryhcode = de2bi(hcode);
19 encodedLen = numel(binaryhcode)
20
Activate Windows
Go to Settings to activate Windows.
line: 20 col: 1 encoding: SYSTEM (CP1252) eol: CRLF
ENG 9:06 PM 3/9/2021


Octave
File Edit Debug Window Help News
Current Directory: folder-20210307T050920Z-001\imagesfolder
File Browser
F:/Surekha/Msc IT Prct/IP Prct/imagesfolder-20210307T050920Z-001/imagesfolder
Name
lena.png
livingroom.tif
logT.m
lowpass_average.m
lowpass_wt_average.m
mandril_color.tif
mandril_gray.tif
Workspace
Filter
Name Class Dimension Value Attribute
A double 248x302 [0, 0, 0, 0, 0, 0, 0...
B uint8 246x300 [49, 50, 48, 49, 4...
F1 double 3x3 [0, 2, 0; 2, -8, 2; ...
I uint8 246x300 [0, 0, 0, 0, 0, 0, 0...
I1 uint8 246x300 [49, 50, 48, 49, 4...
I2 double 246x300 [0, 0, 0, 0, 0, 0, 0...
Command History
Filter
Practical2b
Practical2c
practical3a
practical3b
practical4a
practical4a
Practical7a
Practical7a
Command Window
r2 = 380
c1 = 1
c2 = 445
r1 = 2
r2 = 380
c1 = 2
c2 = 446
r1 = 2
r2 = 380
c1 = 3
c2 = 447
r1 = 3
r2 = 381
c1 = 1
c2 = 445
r1 = 3
r2 = 381
c1 = 2
c2 = 446
r1 = 3
r2 = 381
c1 = 3
c2 = 447
>> practical4a
>> Practical7a
ans = 1
seqLen = 1000
encodedLen = 600
>> Practical7a
ans = 1
seqLen = 1000
encodedLen = 600
>>
Activate Windows
Go to Settings to activate Windows.
Command Window Documentation Variable Editor
ENG 9:07 PM 3/9/2021

Roll No: 12

Practical :-7B

Add a watermark to the image

A digital watermark is a kind of marker covertly embedded in a noise-tolerant signal such as audio, video or image data. ... "Watermarking" is the process of hiding digital information in a carrier signal; the hidden information should, but does not need to, contain a relation to the carrier signal.

Watermark Image

Originally a watermark is a more or less transparent image or text that has been applied to a piece of paper, another image to either protect the original image, or to make it harder to copy the item e.g. money watermarks or stamp watermarks.

There are two types of digital watermarking,visible and invisible.

A visible watermark on a file or image is very similar to a corporation's logo on its letterhead. It is basically a semi- transparent identifier (i.e. logo) that is used to show the ownership of the file or image.

Watermarking Code:

```
pkg load image;

clear all;

close all;

#Input Image where we want to apply watermark

f=imread('lena_color_512.tif');


#For watermarking, size of inputimage and watermarking image should be same

#there for we changed the size of image using imresize and dispalyed


fr=imresize(f,[560 560]);

figure;imshow(fr);

title('Original Image with resized');


#Watermarking Image

w=imread('watersample.jpg');

#Again Resized the Watermarking Image

wr=imresize(w,[560 560]);

figure;imshow(wr);

title('watermark');
```

#Applied watermarking

alpha=0.7;

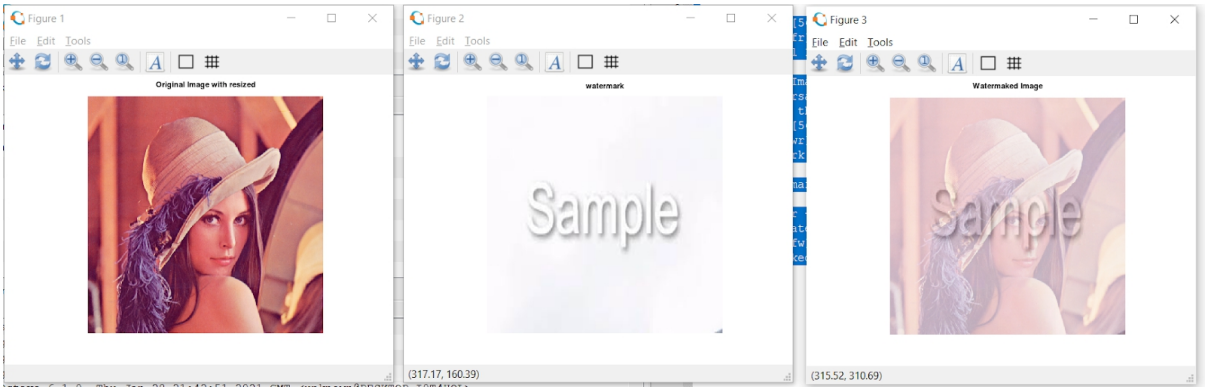
fw=(1-alpha)*fr + alpha.*wr;

#Display the watermarked Image

figure;imshow(fw);

title('Watermaked Image');

Output:



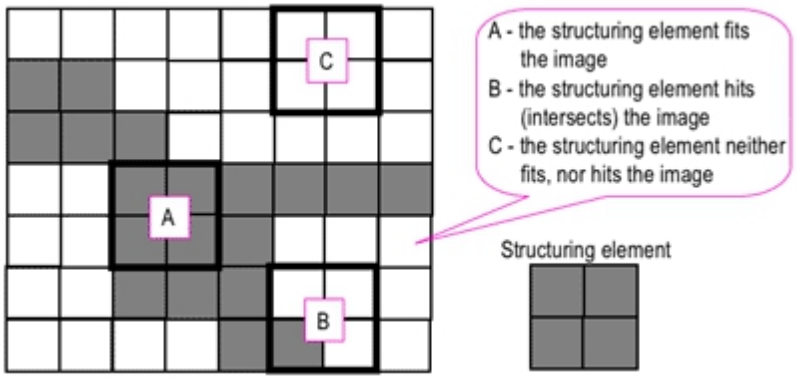
Roll No: 12

Practical No :-8

Morphology

Morphological operators often take a binary image and a structuring element as input and combine them using a set operator. They process objects in the input image based on characteristics of its shape, which are encoded in the structuring element. The mathematical details are explained in Mathematical Morphology.

Usually, the structuring element is sized 3×3 and has its origin at the center pixel. It is shifted over the image and at each pixel of the image its elements are compared with the set of the underlying pixels. If the two sets of elements match the condition defined by the set operator (*e.g.* if the set of pixels in the structuring element is a subset of the underlying image pixels), the pixel underneath the origin of the structuring element is set to a pre-defined value (0 or 1 for binary images). A morphological operator is therefore defined by its structuring element and the applied set operator.



When a structuring element is placed in a binary image, each of its pixels is associated with the corresponding pixel of the neighbourhood under the structuring element. The structuring element is said to **fit** the image if, for each of its pixels set to 1, the corresponding image pixel is also 1. Similarly, a structuring element is said to **hit**, or intersect, an image if, at least for one of its pixels set to 1 the corresponding image pixel is also 1.

$$s_1 = \begin{pmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{pmatrix} \qquad s_2 = \begin{pmatrix} 0 & 1 & 0 \\ 1 & 1 & 1 \\ 0 & 1 & 0 \end{pmatrix}$$

		A	B	C
fit	s_1	yes	no	no
	s_2	yes	yes	no
hit	s_1	yes	yes	yes
	s_2	yes	yes	no

Types of Morphological Operations

Morphological Dilation and Erosion

The most basic morphological operations are dilation and erosion. Dilation adds pixels to the boundaries of objects in an image, while erosion removes pixels on object boundaries.

The number of pixels added or removed from the objects in an image depends on the size and shape of the *structuring element* used to process the image.

Roll No: 12

In the morphological dilation and erosion operations, the state of any given pixel in the output image is determined by applying a rule to the corresponding pixel and its neighbors in the input image. The rule used to process the pixels defines the operation as a dilation or an erosion. This table lists the rules for both dilation and erosion.

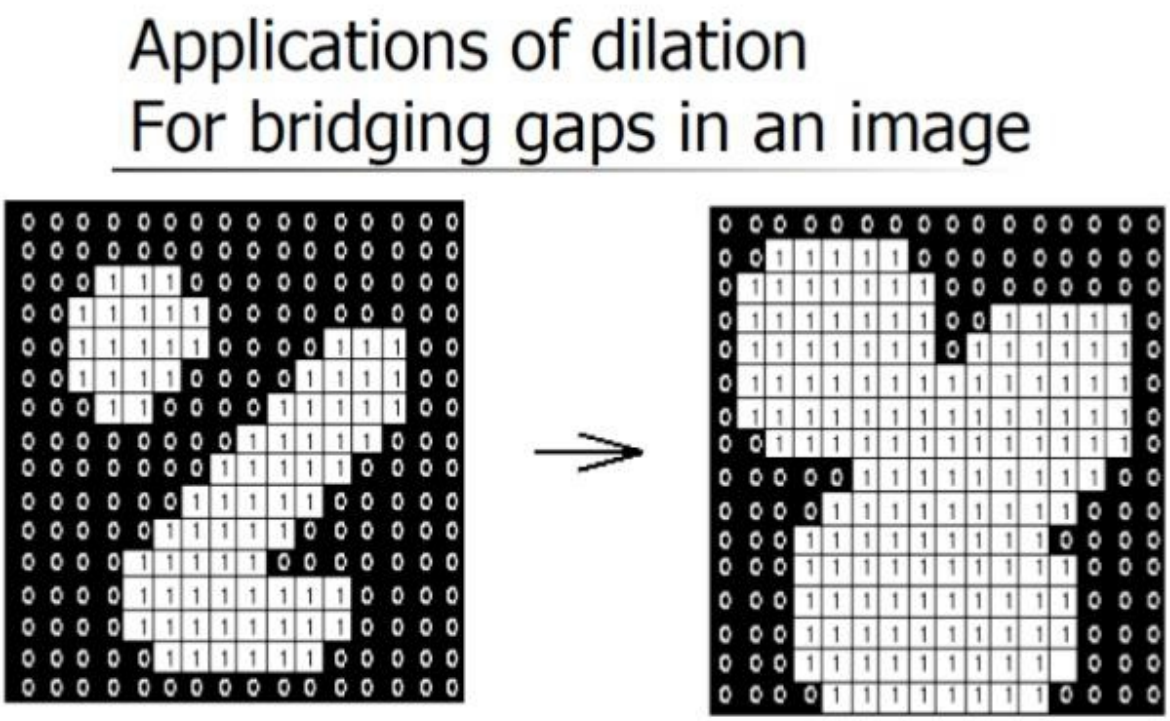
[8.1]Dilation

It grows or thicken object in binary image

Thickening is controlled by a shape referred to as structuring elements

Structuring element is a matrix of 1‟s and 0‟s

A*B=(A+B)



Syntax :

Y=imdilate(A,B)

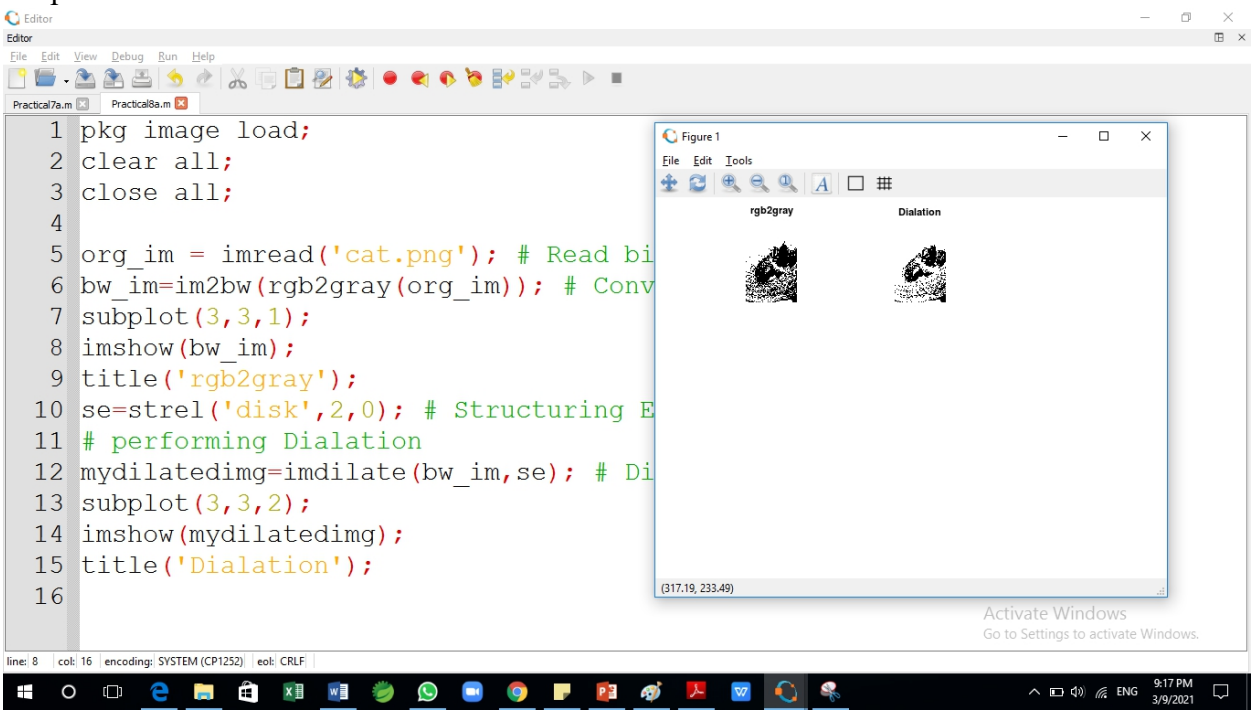
A ->Input image , B ->Structuring elements and Y-> Dilated image

Code :-

```
pkg image load;

clear all;

close all;
```

```
org_im = imread('cat.png'); # Read binary image
bw_im=im2bw(rgb2gray(org_im)); # Convert rgb image to grayscale
subplot(3,3,1);
imshow(bw_im);
title('rgb2gray');
se=strel('disk',2,0); # Structuring Elements (Shape and parameter)
# performing Dialation
mydilatedimg=imdilate(bw_im,se); # Dilated the image
subplot(3,3,2);
imshow(mydilatedimg);
title('Dialation');
```

Output :-

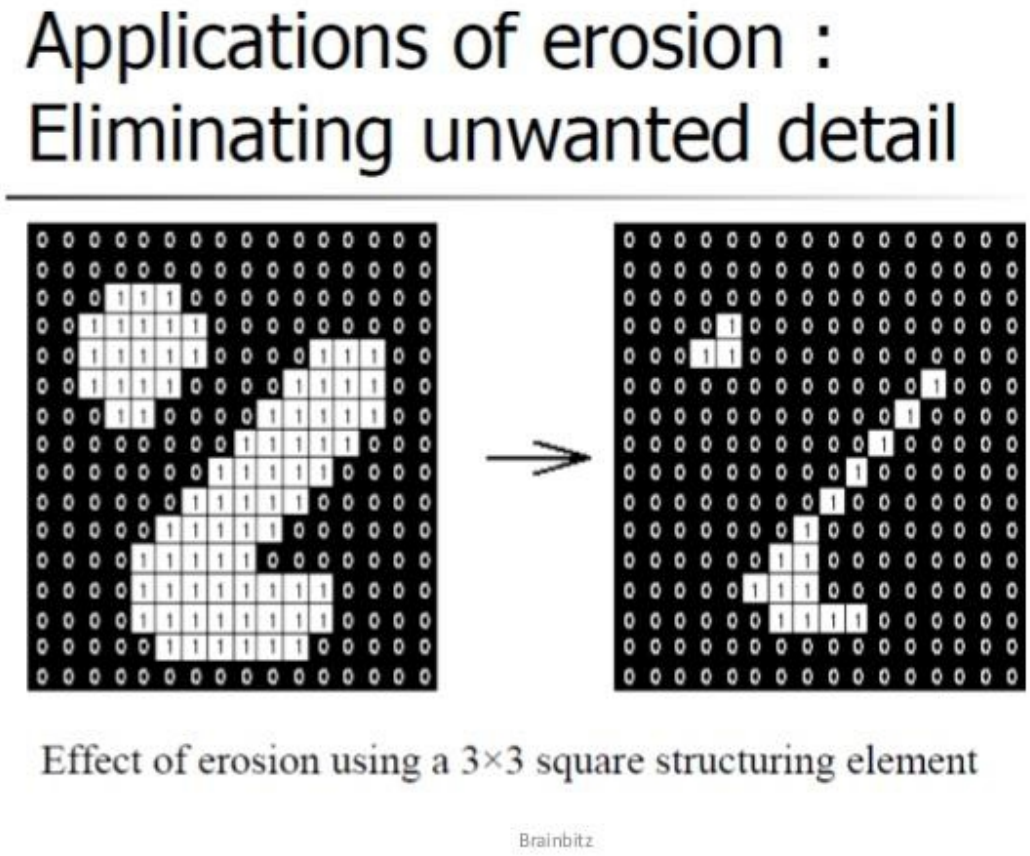


Roll No: 12

Practical 8.2:Erosion

Erosion removes pixels on object boundaries. The number of pixels added or removed from the objects in an image depends on the size and shape of the structuring element used to process the image.

It shrinks or thin object in binary image
A*B=A-B

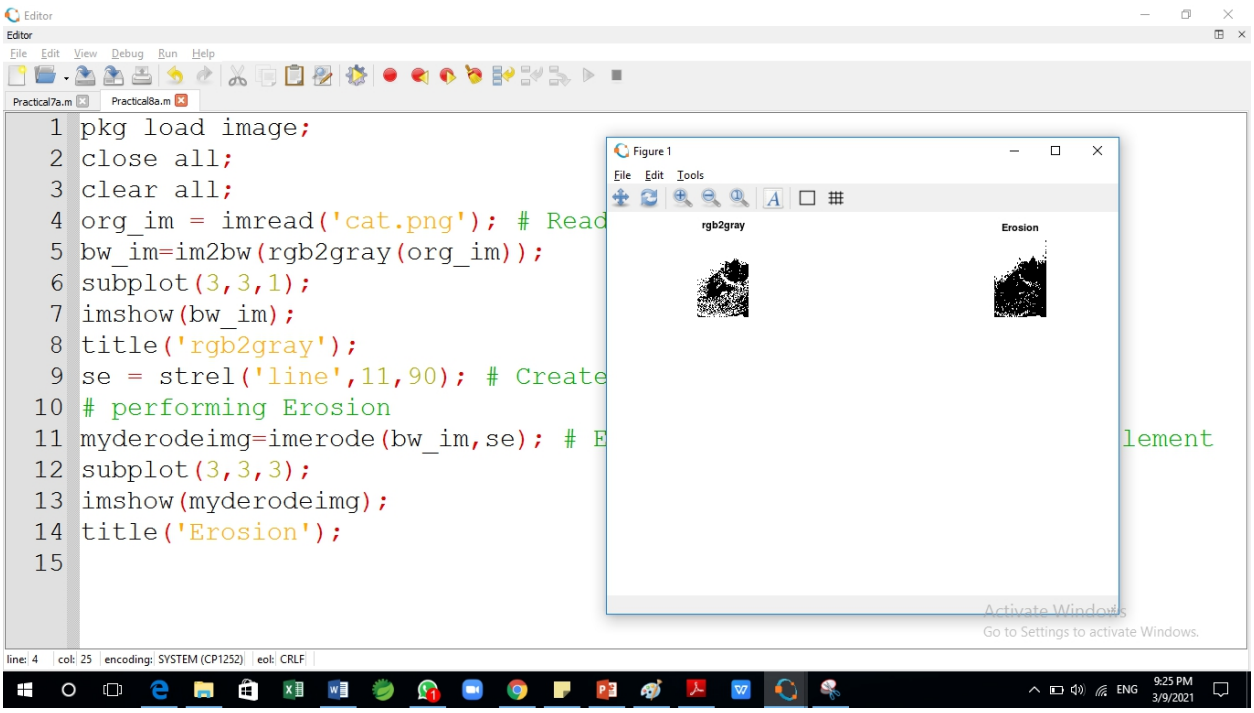


Syntax :-
Se=imerode(A,B)
Code :-

```
org_im = imread(cat.png); # Read binary image
bw_im=im2bw(rgb2gray(org_im));
subplot(3,3,1);
imshow(bw_im);
title('rgb2gray');
se = strel('line',11,90); # Create flat line structuring element
# performing Erosion
```

```
myderodeimg=imerode(bw_im,se); # Erode the Image with structuring element
subplot(3,3,3);

imshow(myderodeimg);
title('Erosion');
```

Output :-



Roll No: 12

Practical 8.3-Boundary Extraction

An edge in an image is a boundary or contour at which a significant change occurs in some physical aspect of an image, such as the surface reflectance, illumination or the distances of the visible surfaces from the viewer.

Internal Boundary

Algorithm for Boundary Extraction

Dilation Algorithm: The boundary of a set A, denoted by $\beta(A)$, can be obtained by first dilating A by B and then performing the set differences between A and its dilation. That is,

$\beta(A) = (A \; B) - A$

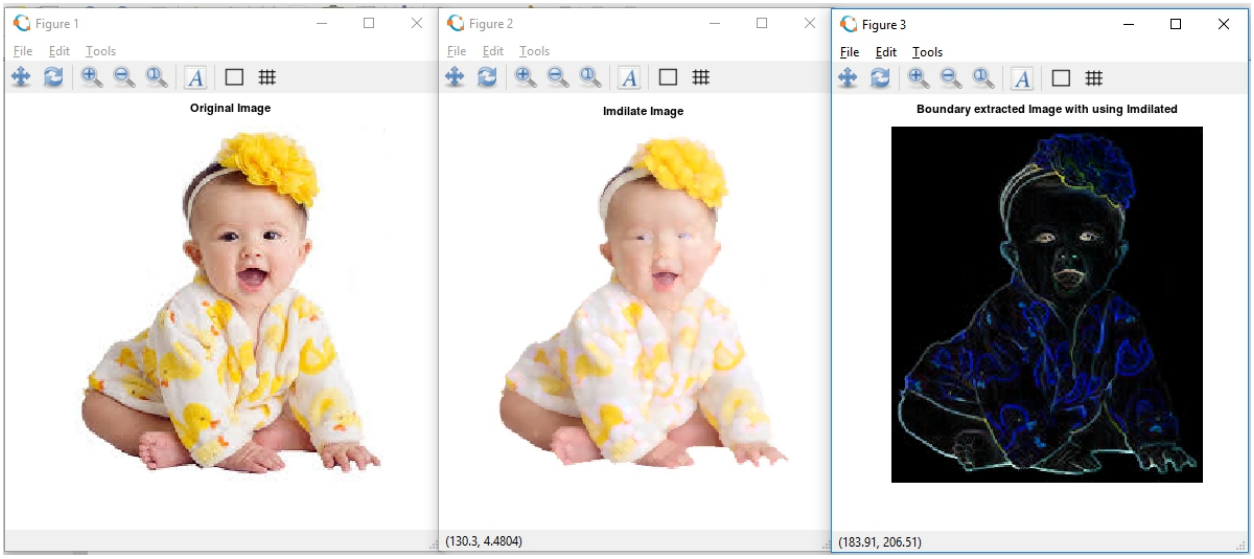
Boundary Extraction with help of Dilation

Code :-

```
pkg load image

A=imread('baby.jpg');

s=strel('disk',2,0); # Structuring element shape and parameter

F=imdilate(A,s); # Dilate the image by structuring elements

figure

imshow(A);

title('Original Image');

figure

imshow(F);

title('Imdilate Image');

figure

imshow(F-A);

title('Boundary extracted Image with using Imdilated');
```

```
pkg load image
A=imread('baby.jpg');
s=strel('disk',2,0); # Structuring element shape and parameter
F=imdilate(A,s); # Dilate the image by structuring elements
figure
imshow(A);
title('Original Image');
figure
imshow(F);
title('Imdilate Image');
figure
imshow(F-A);
title('Boundary extracted Image with using Imdilated');
```

Output :-
Original Image



Roll No: 12

External Boundary :-

Algorithm for Boundary Extraction

- **Erosion Algorithm:** The boundary of a set A, denoted by $\beta(A)$, can be obtained by first eroding A by B and then performing the set differences between A and its erosion. That is,

  $\beta(A)=A - (A \ominus B)$

Boundary Extraction

with help of erosion

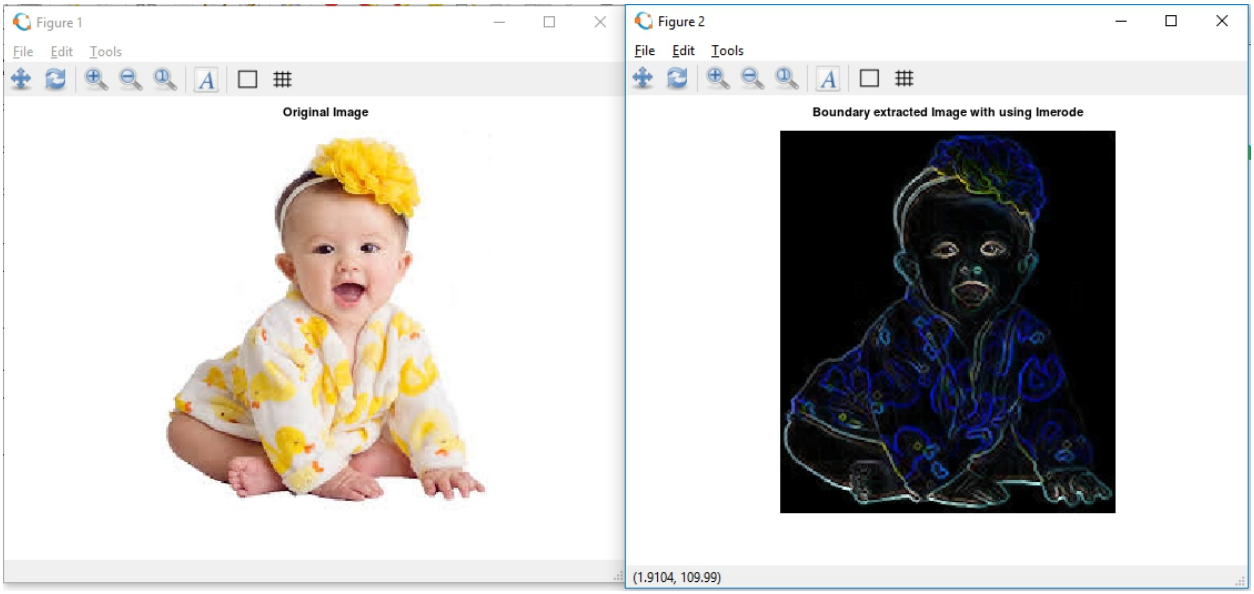
Code :-

```
pkg load image

A=imread('baby.jpg');

s=strel('disk',2,0); # Structuring element shape and parameter

F=imerode(A,s); # Erode the image by structuring elements

figure

imshow(A);

title('Original Image');

figure

imshow(A-F);

title('Boundary extracted Image with using Imerode');
```

```
pkg load image
A=imread('baby.jpg');
s=strel('disk',2,0); # Structuring element shape and parameter
F=imerode(A,s); # Erode the image by structuring elements
figure
imshow(A);
title('Original Image');
figure
imshow(A-F);
title('Boundary extracted Image with using Imerode');
```

Output :-

Original Image and Boundary extracted image using imerode:-



Roll No: 12

[8.4]Skeleton Image :-

Skeletonization is a process for reducing foreground regions in a binary image to a skeletal remnant that largely preserves the extent and connectivity of the original region while throwing away most of the original foreground pixels.

The skeleton is thin. It contains much less points than the original elongated object

Skeletonization (i.e., skeleton extraction from a digital binary picture) provides region-based shape features. It is a common preprocessing operation in raster-to-vector conversion or in pattern recognition.

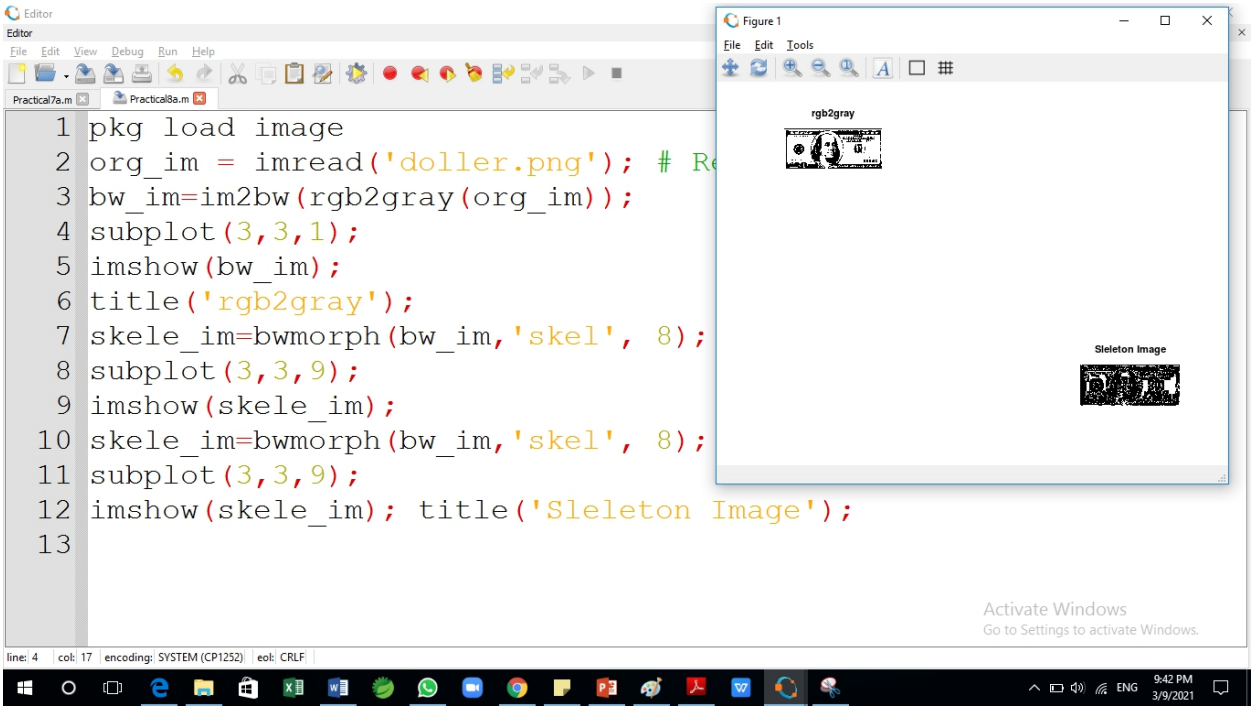
There are three major skeletonization techniques:

- detecting ridges in distance map of the boundary points,
- calculating the Voronoi diagram generated by the boundary points, and
- the layer by layer erosion called thinning.

Code:-

```
pkg load image

org_im = imread('doller.png'); # Read binary image

bw_im=im2bw(rgb2gray(org_im));

subplot(3,3,1);

imshow(bw_im);

title('rgb2gray');

skele_im=bwmorph(bw_im,'skel', 8);

subplot(3,3,9);

imshow(skele_im);

skele_im=bwmorph(bw_im,'skel', 8);

subplot(3,3,9);

imshow(skele_im); title('Sleleton Image');
```

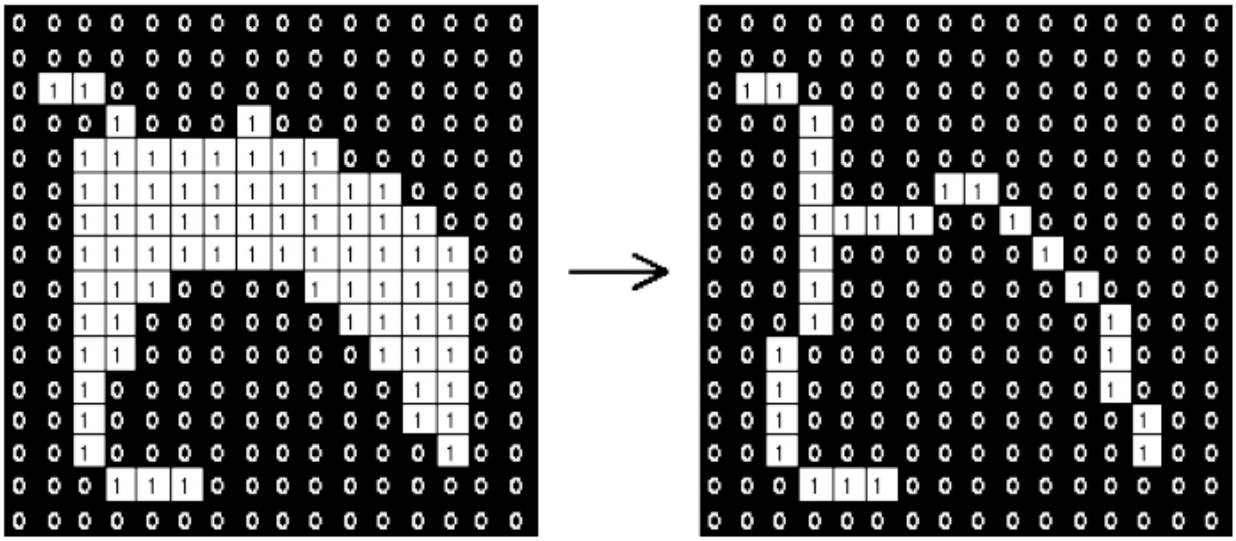
Output :-



Roll No: 12

Practical 8.5-Thinning

Thinning is a morphological operation that is used to remove selected foreground pixels from binary images, somewhat like erosion or opening. It can be used for several applications, but is particularly useful for skeletonization.

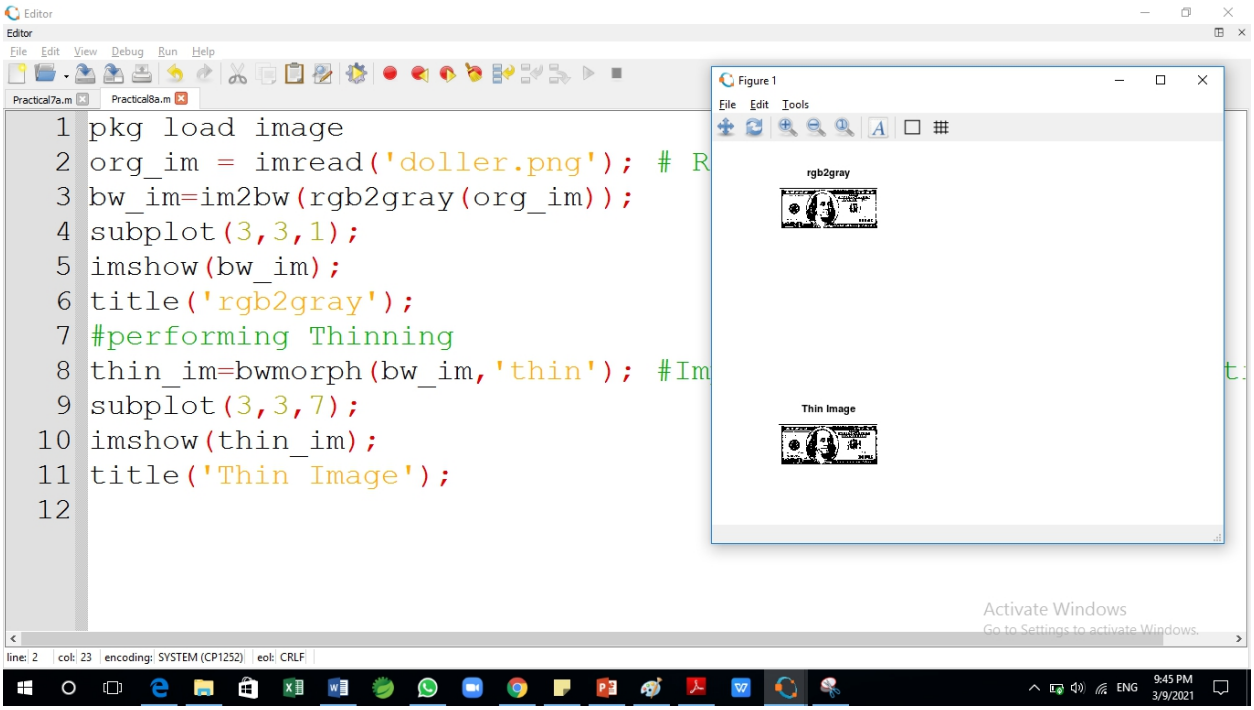


In this mode it is commonly used to tidy up the output of edge detectors by reducing all lines to single pixel thickness [6].
Like other morphological operators, thinning operators take two pieces of data as input. One is the input image, which maybe either binary or greyscale. The other is the structuring element, which determines the precise details of the effect of the operator on the image.

Code :-

```
pkg load image

org_im = imread('doller.png'); # Read binary image

bw_im=im2bw(rgb2gray(org_im));

subplot(3,3,1);

imshow(bw_im);

title('rgb2gray');

#performing Thinning

thin_im=bwmorph(bw_im,'thin'); #Implements morphological operation

subplot(3,3,7);

imshow(thin_im);

title('Thin Image');
```

Output :-



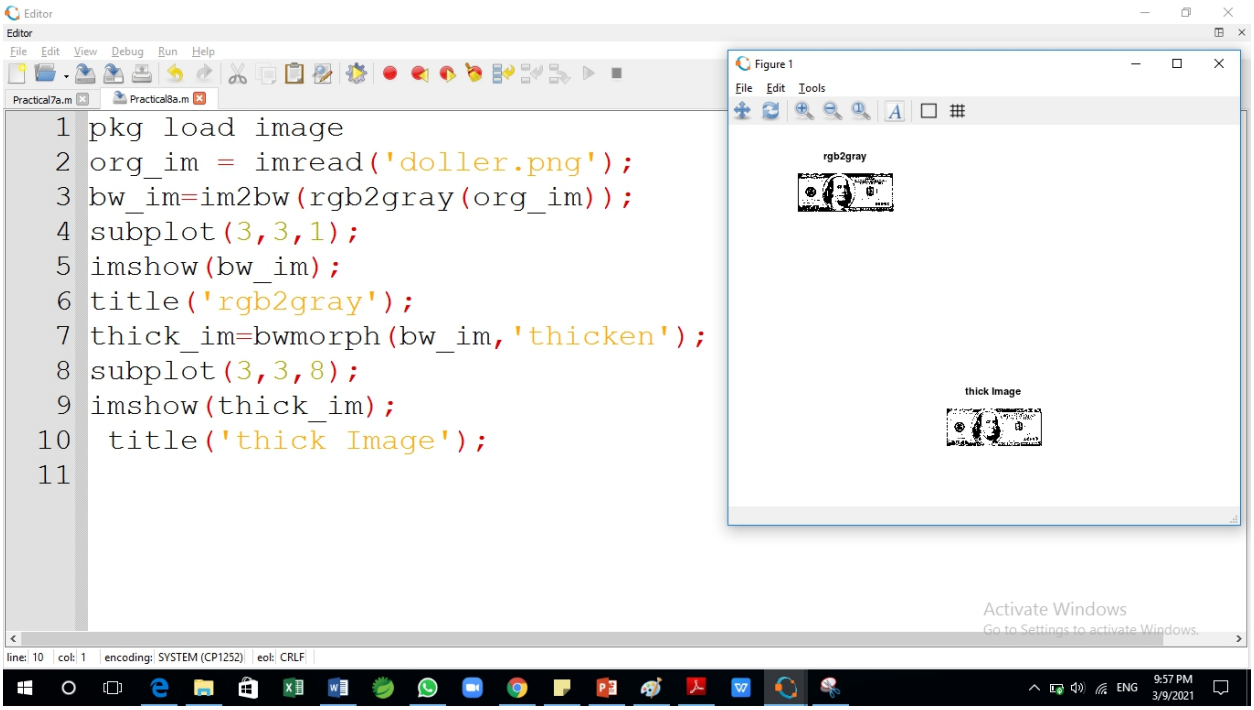
[8.6]Thickening

Thickening is a morphological operation that is used to grow selected regions of foreground pixels in binary images, somewhat like dilation or closing. ... Thickening is normally only applied to binary images, and it produces another binary image as output.

Code :-

```
pkg load image
org_im = imread('E:\Omkar\oneplus.jpg');
bw_im=im2bw(rgb2gray(org_im));
subplot(3,3,1);
imshow(bw_im);
title('rgb2gray');
thick_im=bwmorph(bw_im,'thicken');
subplot(3,3,8);
imshow(thick_im); title('thick Image');
```

Output:



Roll No: 12